DOCUMENT RESUME

ED 286 479 IR 012 812

AUTHOR	Heimler, Charles; And Others
TITLE	Authoring Educational Software.
REPORT NO	ISBN-0-394-39051-2
PUB DATE	87
NOTE	276p.
AVAILABLE FROM	Mitchell Publishing, Inc., 915 River Street, Santa Cruz, CA 95060 ($20.00).
PUB TYPE	Books (010) -- Guides - Classroom Use - Guides (For Teachers) (052)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Authoring Aids (Programing); *Computer Assisted Instruction; *Courseware; Elementary Secondary Education; Higher Education; *Programing Languages
IDENTIFIERS	Applesoft BASIC; *Screen Format; *Software Design

ABSTRACT

Intended for use by preservice and in-service teachers, this book contains directions for development of Applesoft BASIC programs that provide instruction related to the learning of school subjects. Chapter 1 tells how to select a topic suitable for educational software and gives directions for developing program content based on specified learning objectives. Applesoft BASIC is reviewed in Chapter 2, and chapter 3 shows how to develop a program as a series of frames with interesting screen displays for selected frames. Chapter 4 explains how to make an educational program interactive through the use of prompts and input statements while chapter 5 describes procedures for including evaluation of student achievement. Different routines to add variety and interest to a program are presented in chapter 6, and chapter 7 contains short routines to display and format numbers. Chapter 8 describes shortcuts for debugging a program, and chapter 9 explains how to write documentation for the program. The appendices include sample programs, a list of criteria for the design and selection of educational software, and a table of ASCII character codes. (MES)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

BEST COPY AVAILABLE

ED286479

authoring educational software

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

IR012812

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY

Raleigh Wilson

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

AUTHORING EDUCATIONAL SOFTWARE

Charles Heimler
James Cunningham
Michael Nevard

Mitchell Publishing, Inc.
Innovators in Computer Education
915 River Street • Santa Cruz, California 95060
(800) 435-2665 • In California (408) 425-3851

Copyright © 1987
Mitchell Publishing, Inc., a Division of Random House, Inc., New York City
Innovators in Computer Education
915 River Street
Santa Cruz, CA 95060
(800) 435-2665; (408) 425-3851 in California

All rights reserved.
The text of this publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without prior permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Library of Congress Catalog Card Number: 86-062003

ISBN: 0-394-39051-2

Sponsoring Editor: Raleigh Wilson
Production Management: Greg Hubit Bookworks
Cover Illustration: Henry J. Breuer
Text and Cover Design: Rodelinde Albrecht

CONTENTS

ONE EDUCATIONAL PROGRAMS 1

1.1 What Is an Educational Program? 2
1.2 Selecting the Topic 5
1.3 Examples of Topics 7
1.4 How to Write an Educational Computer Program 10
1.5 Learning Objectives 18
1.6 Criteria for Design and Development 22
1.7 Structured Programming 25
1.8 Programming the Content 31

TWO A SHORT REVIEW OF APPLESOFT BASIC 39

2.1 Variables 40
2.2 READ/DATA 43
2.3 INPUT and GET Statements 46
2.4 Branching 48
2.5 Multiple Statement Lines 52
2.6 FOR/NEXT 54
2.7 Subscripted Variables 57
2.8 String Functions 61
2.9 Concatenation 64
2.10 CHR$ and ASC 65
2.11 LEN 66
2.12 VAL and STR$ 67

2.13 ON GOTO and ON GOSUB 68
2.14 PEEK and POKE 69

THREE SCREEN FORMAT 73

3.1 Frames 74
3.2 Screen Displays 78
3.3 Screen Formatting CALLs 81
3.4 Delay Loop 86
3.5 Text on Screen 89
3.6 Text Window 92
3.7 Screen Borders 96
3.8 Input Question Mark 98

FOUR INTERACTIVE 101

4.1 Interactive Is Important 102
4.2 User Control 106
4.3 Prompts 109
4.4 Inputs 114
4.5 Input Checks 116
4.6 Multiple Choice GET 121
4.7 String Search 123
4.8 Keyboard Control 129
4.9 Programming a Menu 132

FIVE STUDENT ACHIEVEMENT 139

5.1 Reinforcements 140
5.2 Counters and Tallies 143
5.3 Counters and Tallies in Subroutines 146
5.4 Counting Entries 148
5.5 Evaluation of Student Achievement 150

SIX VARIETY AND INTEREST 157

6.1 Text Motion 158
6.2 Spinner 162
6.3 Speed Change 164
6.4 Random Function 167

6.5 Sorting Routines 172
6.6 Digital Counter 179
6.7 Character Graphics 183

SEVEN NUMERIC OUTPUT 191

7.1 Commas in Numerals 192
7.2 Scientific Notation Conversion 195
7.3 Dollars and Cents 197
7.4 Right Justification 200
7.5 Decimal Point Alignment 203

EIGHT DEBUGGING 209

8.1 STOP and CONTinue 210
8.2 Tracing Line Numbers 215
8.3 READ/DATA and RESTORE 220
8.4 ONERR GOTO 226

NINE DOCUMENTATION 233

9.1 Why Document? 234
9.2 What to Document 236
9.3 Format for Documentation 237

APPENDICES

A The Number Machine 243
B The Number Machine Worksheet 247
C End Punctuation 248
D Animal Heartbeats 255
E Criteria for the Design and Selection of Educational Software 259
F Table of ASCII Character Codes 261

INDEX 263

PREFACE

This is a book for preservice and in-service teachers interested in the creation of educational software for use by students. It contains specific and concise directions for the development of Applesoft BASIC programs that provide instruction related to the learning of school subjects.

FEATURES

- Explains how to select a topic for an educational program and how to write the program so that the final product will be interesting and meaningful to the students who use it. Short program listings are included that show how to add novelty, variety, and interesting screen displays to a program. Techniques for programming learning objectives, user friendliness, keyboard control, reinforcements, animation, character graphics, and digital counters are presented.
- Includes unique programming routines that can be used by anyone with a knowledge of BASIC. Included, for example, are routines that show how to print borders on the screen with keyboard characters and the use of the GET statement to provide user control in running a program. Directions for creating menus, trapping input errors, and using random numbers to increase a program's flexibility are provided. Three programs are presented that illustrate the writing of

an educational computer program: Number Machine, Animal Heartbeats, and End Punctuation. A complete listing of each program is given in the Appendices, and routines from the three programs are used periodically in the text to illustrate programming techniques.

- Presents, for the first time together, educational programming techniques and routines that explain through examples and illustrations exactly how they are used in writing an educational program. For each program routine, the line statements are listed and the execution of each statement is discussed. Simplified directions show how to place each routine in a program so it will run correctly. All routines are written in Applesoft BASIC, since the Apple II series of microcomputers that run this version of BASIC are available in an ever-increasing number of schools and colleges. Moreover, many teachers have learned or are learning to program in Applesoft BASIC. The programs and routines may be easily modified so that they can be programmed on the IBM PC, TRS-80, Commodore, and other microcomputers.

ORGANIZATION

- Chapter 1 tells you how to select a topic suitable for an educational computer program. Directions are given for developing the content of a structured program based on specified learning objectives.
- Chapter 2 is a review of Applesoft BASIC with emphasis on aspects of this language particularly pertinent to educational computer programming.
- Chapter 3 shows you how to develop a program as a series of frames and explains how you can create interesting screen displays for selected frames within a program.
- Chapter 4 explains how to make an educational program interactive through the use of prompts and input statements. Routines for input checks are also discussed in this chapter.
- Chapter 5 describes procedures you can use for including the evaluation of student achievement. Directions are given for programming reinforcements, counters, and scoring correct responses.
- Chapter 6 presents different routines you can include in a program that add variety and interest—for example,

routines for moving text, speeding changes in the printing of text, and including a digital clock in a program.
- Chapter 7 contains short routines you can include in a program to display and format numbers on the computer screen—for example, adding commas to numerals, printing dollars and cents, and aligning decimal points in columns of numbers.
- Chapter 8 describes shortcuts you can use to debug a program as you develop the final product.
- Chapter 9 explains how to write a short documentation that delineates the scope and content of an educational program.

FOCUS

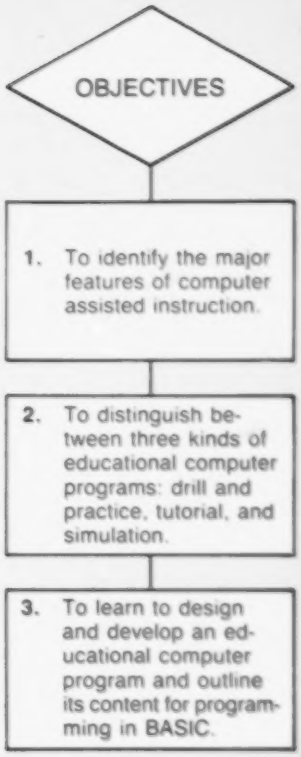
Computers and related educational software are rapidly becoming an integral part of teaching and learning in America's schools and colleges as smaller, cheaper, and faster microcomputers continue to enter the learning environment in ever increasing numbers. Foremost in educational computing is the fact that students like to use computers; they quickly become interested in a program that helps them learn the material they are covering in class. A properly designed program allows the student to interact with the computer by entering responses to directions and questions and provides immediate feedback, based on these entries, that guides the student's learning. Moreover, use of an educational computer program provides a personalized student-oriented learning environment, allowing the student an opportunity to self-pace the rate at which material is presented and, in some cases, to select the level of difficulty at which the content is presented.

"How to" directions presented in this book will be of great assistance to all those who are now writing or planning to write educational programs in BASIC. Computers and related software will eventually revolutionize the ways in which people teach and learn, from kindergarten through the university level. It is the intention of the authors in their writing to impart to the reader some measure of the fun, excitement, and feeling of accomplishment associated with the development and use of educational computer programs and to provide practical, concrete assistance in the creation of such programs.

AUTHORING EDUCATIONAL SOFTWARE

CHAPTER ONE

EDUCATIONAL PROGRAMS


OBJECTIVES
1. To identify the major features of computer assisted instruction.
2. To distinguish between three kinds of educational computer programs: drill and practice, tutorial, and simulation.
3. To learn to design and develop an educational computer program and outline its content for programming in BASIC.

In this chapter you will learn the essentials of computer assisted instruction and its implications for modern education. The main features that distinguish three kinds of educational computer programs—drill and practice, tutorial, and simulation—are identified and discussed. Specific criteria are given to guide you in selecting a topic that can be developed into an educational computer program. Examples in language and literature, social studies, mathematics, and science are given to illustrate suitable topics for educational computer programming. Sections 1.5 and 1.6 focus on using learning objectives to select and develop the content for an educational computer program to achieve these objectives. Structuring and writing the content of a program in BASIC are covered in Sections 1.7 and 1.8.

1.1 What Is an Educational Program?

An educational computer program presents knowledge or skills that can be learned by someone using a computer to run the program. For example, a student may use a program to learn elementary arithmetic, or a student can be introduced to the basic business principles involved in managing a lemonade stand by running a program that calls for pricing a glass of lemonade, deciding how much lemonade to make, and how many and what type of advertising signs to display for different weather conditions. Three Mile Island is a program in which students learn how the parts of a nuclear reactor interact to produce energy and how the reactor safety mechanisms are controlled to prevent overheating and meltdown. Each program, like other educational programs, focuses on teaching facts and ideas that are part of the program's content.

Educational computer programs may be categorized as drill and practice, tutorial, and simulation. Drill and practice programs are those in which the student performs some activity repetitively: for example, drill and practice in spelling, multiplication tables, and English grammar. The repeated practice provided by the program helps the student memorize certain information or develop a skill. Programs providing drill and practice are often designed as games that use vivid color graphics to motivate interest. Drill and practice normally requires no more than rote learning or simple memorization, because little or no conceptual learning need be involved. However, programs of this type do furnish students the opportunity to work on material at their level at a self-paced rate for as long as they desire. Drill and practice programs free the teacher from

spending class time on this activity. See Appendixes A and B for The Number Machine, a drill and practice program designed to be used with a worksheet.

The tutorial category covers those programs that present information for the student to learn by reading and answering questions related to the information. Tutorial programs are the closest to textbook learning. Associating correctly the names of classical authors with quotations from their writings is an example of one kind of tutorial. Another type of tutorial is Socratic in style. As a Socratic program runs, the student engages in active dialogue, interacting with sequences of questions and answers posed by the computer. Appendix C lists End Punctuation, a tutorial program that teaches sentence end punctuation.

Simulation programs are those that present a representation of some event or phenomenon. For example, the program may simulate the real-life effects of pollutants on the ecology of a lake. By adding and removing pollutants from the water during the run of the program, the student learns how these changes affect the health of the plants and animals in the lake. Many natural events lend themselves to modeling and are therefore suitable for computer simulation. Educationally, the computer simulation provides experience a student might not be able to obtain firsthand. Computer simulations include programs portraying earthquakes, volcanic eruptions, prospecting for oil, and flying an airplane. Through such programs, a student gains valuable insights and understandings not generally available through more conventional methods of study.

As with other methods of instruction, interaction is important in learning with computers. When a student runs an educational program he or she interacts with information presented on the video screen. This interaction may take the form of reading and answering questions pertaining to the information. The student may also interact by calculating and inputting answers to math problems. Immediate feedback is provided to the student as the computer indicates whether each answer is right or wrong. If an answer is incorrect, the program prints the correct answer and, for a math problem, shows how to calculate the answer correctly.

Some educational computer programs teach a concept by presenting a phenomenon the student must analyze. In a program called Animal Heartbeats, the repeated beeps of the computer's speaker represent the rate of an animal's heartbeat. (Animal Heartbeats is listed in Appendix D.) By comparing the heartbeat rates for different animals, from guinea pig to elephant, the student learns that the rate varies inversely with the animal's weight. In this program, as with all other educational computer programs, interaction,

the give and take between the student and the computer, is the essence of the learning process.

Educational computer programs do not preclude the use of lectures, discussions, textbooks, laboratory activities, and other conventional means of instruction. The program, as with other methods of teaching, assist students in learning. Computer assisted instruction (CAI) is gaining increasing use in education. Simply stated, CAI refers to just about any use of computers in learning. It covers the use of educational computer programs such as those we have described as well as other uses of a computer that a student might make in or out of school in learning knowledge or skills.

Although not originally designed for educational purposes, word processing programs, electronic spreadsheets, and database programs can be used in teaching and learning. Bank Street Writer and Home Word are examples of word processing programs a student can use to learn how to write an outline or report for an English class. An electronic spreadsheet is used in business classes. With this kind of program the student enters numerical values for the different parts of a financial plan and uses the computer to change various aspects of the plan by speedily recalculating the values. One illustration of the educational use of an electronic spreadsheet is planning and analyzing a budget for a small business. Database programs are used in a variety of subjects, including social studies, home economics, and science, to teach students how to categorize, store, and retrieve information. Educational leaders recognize that word processing programs, electronic spreadsheets, and database programs are three important computer tools in today's commercial world, and they advocate use of such programs in education.

Educational computer programs are productive because their use in learning is supported by several principles of educational psychology. Foremost is that the computer provides immediate feedback. There is no waiting for a teacher to grade a set of papers. Computer learning is individualized in the sense that, when using the computer, students can control the pace at which the program proceeds and may rerun a program as often as they like. Options within a program let students choose the content and instructional level. Therefore most students like to use computers, and their use can increase the time they spend studying a topic. According to the time on task principle, learning achievement correlates positively with the amount of time a student spends in the learning process. Therefore, as teachers and students become more aware of the motivation and achievement to be gained by using computers, this form of instruction, and the related development of educational computer programs, will claim an ever-increasing presence in the teaching-learning process of education.

1 What are the main features of an educational computer program?

2 Distinguish between drill and practice, tutorial, and simulation programs.

3 Describe two examples of each kind of educational computer program.

4 How can each of the following kinds of computer programs help students learn: word processing, electronic spreadsheet, database?

5 How can educational computer programs be used in your teaching field?

Selecting the Topic 1.2

A major factor in creating an educational computer program is choosing a topic for which the program will provide instruction. It is best to choose a topic that takes advantage of the special capabilities of the computer. Generally, material involving logic, choices, arrangements, comparisons, numbers, or calculations falls into this category and may be developed into a worthwhile program. Examples of topics having one or more of these characteristics include making change from a dollar bill, symbols for chemical elements, states and capitals, color coding electrical resistors, and standard English grammar.

The three programs in the Appendixes illustrate topics particularly suited to educational programming. Number Machine provides drill and practice in writing numbers as words. A student using the program types in a number, and the computer prints the number literally on the screen. For example, the student enters 2001, and the computer prints TWO THOUSAND ONE. Animal Heartbeats simulates the heartbeat rates of five different animals to show the relationship between an animal's heartbeat rate and mass. End Punctuation is a tutorial providing instruction in the punctuation used at the end of a sentence.

We recommend that you select a topic you can develop into a short, succinct program having maximum learning value. Do not

select a topic that is too broad and unwieldy, for it will likely result in a program that is uneven and drawn out. Such a program will not hold a student's interest. Don't forget: the attention span of children is not as long as it is for adults. It is better to have a tight program interesting enough to motivate a student to run it several times than to have a long program that engenders boredom and disinterest.

Choose a specific skill or concept that is elucidated through a program focusing on learning the main idea involved. For example, converting customary units to metric units, comparing the relative caloric values of foods, and recognizing the difference between a noun and a verb. Good sources for locating topics for educational programs are textbooks, workbooks, laboratory manuals, and other activity books. You may wish to choose material such as multiplication tables for which learning requires repeated drill and practice. In some cases you may take a game such as Nim or a classroom learning activity and convert it into a computer program. One example is a program that shows students how to create their own "flashcards" on the computer screen and how to use the cards to review the material they are learning.

Another approach is to create a program that engages a student in learning by studying a simulation. For example, the game Monopoly simulates real-life buying and selling of real estate. Coin tossing and dice rolling, used to teach basic principles of probability, can be simulated through random grapnic presentations that picture these events on a computer screen.

Avoid selecting a descriptive topic that calls for a computer program loaded with lengthy lines of text requiring repeated "page turning" as the user reads through the material. It is much better for the student to use a textbook to learn this kind of information.

It makes sense to choose a topic with which you are familiar. You should already know the material or take the time to learn it before you begin to write the program. An excellent topic to choose for an educational program is one you have already presented through conventional teaching methods, since you are likely to be aware of any pedagogical problems connected with learning it. Above all, particularly if this is your first venture into writing an educational program, do not overextend yourself. Some people with good and sincere intentions have an unusual tendency to obfuscate and convolute the obvious when they try to teach it. So in your selection of a topic and the material involved, keep in mind the ancient acronym KISS—"keep it simple and succinct."

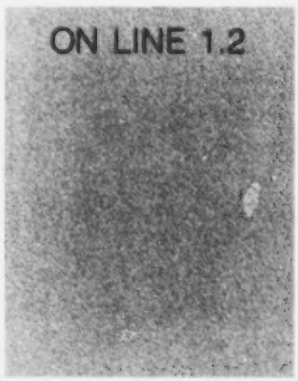
1 What kinds of topics are most suitable for educational computer programs?

2 What are some good sources for locating topics?

3 List five criteria for selecting a topic to develop as an educational computer program.

4 How is the attention span of a student related to the selection of a topic?

5 List five examples of topics in your subject field suitable for educational computer programs.



Examples of Topics 1.3

This section lists and briefly describes examples of topics particularly suited for educational programs. If this is your first venture into writing a program designed to help a student learn something, you might want to select one of these topics. However, rather than being definitive and exclusive, the examples are given to help stimulate your thinking in these areas. Do not hesitate to go beyond this list to select a topic for the program you will design and develop. Remember, your efforts are likely to be most successful if you choose a topic with which you are familiar and that has special interest for you.

LANGUAGE AND LITERATURE

- *Possessive Case.* Use of the apostrophe and the letter *s* to express possessives for singular and plural nouns.
- *Parts of Speech.* Drill and practice in recognizing parts of speech, such as verb, noun, adjective, and adverb.
- *Sentence Combining.* Combining short sentences into longer, grammatically correct sentences.
- *Classics.* Identifying a classic work of literature and its author based upon selected quotations.

- *Punctuation.* Inserting punctuation marks in sentences.
- *Reading.* Adjusting the speed of text presented on the screen and testing reading comprehension.

SOCIAL STUDIES

- *Fact and Judgment.* Learning and applying criteria for distinguishing factual and judgmental statements.
- *States and Capitals.* Drill and practice in identifying the capitals of states.
- *Historical Time Line.* Using the dates of historical events to create historical time lines.
- *World Currencies.* Comparing monetary systems throughout the world and the equivalent exchange rates.
- *Explorers.* Simulating the adventures of a famous explorer, such as Marco Polo, Columbus, or Magellan.
- *Presidents.* Names and important facts pertaining to American presidents.
- *Wars.* Chronology of major American military actions.

MATHEMATICS

- *Computation.* Drill and practice in addition, subtraction, multiplication, or division.
- *Signed Numbers.* Concept of a number line and computation with signed numbers.
- *Coordinates.* Horizontal and vertical coordinates, X,Y axes.
- *Combinations.* Calculating mathematical combinations.
- *Probability.* Mathematical probabilities in simulated events such as coin tossing.
- *Geometry.* Recognizing and comparing the properties of polygons.
- *Statistics.* Calculating the mean, median, and mode.
- *Cash Register.* Making change and calculating sales tax on purchases.

SCIENCE

- *Chemical Elements.* Drill and practice in the symbols and names of chemical elements.

- *Density.* Concept of density and the variation in density among different substances and materials.
- *Symbols and Formulas.* Practice in combining chemical symbols into formulas for chemical compounds.
- *Diffusion and Osmosis.* Tutorial program in which laboratory demonstrations of diffusion and osmosis are described and the student responds to questions pertaining to the demonstrations.
- *Gas Laws.* Effects on gases of changes in pressure, temperature, and volume.
- *Kinds of Matter.* Distinguishing among elements, compounds, and mixtures.
- *Weather Data.* Recording weather data and using the data to understand changes in the weather.
- *Parts of a Cell.* Using models and analogies to teach the parts of a cell and their functions.
- *Minerals.* Identifying minerals based on their physical properties.
- *Predator–Prey.* Simulating a predator–prey relationship, such as the food chain link between sharks and smaller fish. Using the simulation to understand this ecological concept.
- *pH.* Using litmus or another indicator to test the acidity or alkalinity of a solution.
- *Clouds.* Identifying different kinds of cloud formations.
- *Metric System.* Drill and practice in using metric units.
- *Foods.* Basic four food groups.

---

1 Using examples listed in this section, compare and contrast drill and practice programs with tutorial programs.

2 List three examples of suitable topics for development as simulations.

3 Select a topic for development into an educational computer program. List five reasons why you believe the topic is a good choice.

1.4 How to Write an Educational Computer Program

After you have selected a subject and topic for an educational program, it is time to begin planning how to write it. One approach that works well is to follow a teaching model conceptualized in the following four questions:

1. Teach what?
2. Teach whom?
3. Teach how?
4. How well?

This model has a simplicity that belies its power and usefulness in constructing programs designed to teach somebody something. "Teach what?", the first element in the model, means to determine what is to be taught—that is, the content in the presentation. For example, if a program is being developed to teach the properties of vitamins, you must decide what vitamins and what features of these vitamins to cover. A program on vitamins intended for a college biochemistry class would present the individual molecular structures of vitamins as part of the content. However, if the program is intended for a junior high school health class, there is no need to include this information. The emphasis here should be on the food sources of vitamins and their nutritional effects. Thus, "Teach what?" means that you, the programmer, must carefully consider the objectives or purposes of your program and include the content needed to achieve these objectives.

"Teach whom?" means to consider in relation to the material to be taught, the age and background of the student who will use your program. Is the content you have chosen within the learning ability level of the student? Is the material going to be over the student's head? Are the words and terms used in the program likely to be understood? If factors pertaining to the suitability and appropriateness of the content are not taken into account, there will be a problem in communication, and the program, no matter how well put together, will be of little value to the student.

"Teach how?" refers to how you present and develop the material to be learned. Perhaps your program is going to be a drill and practice type in which the student will repetitively respond to questions by entering answers. It may be a tutorial program in which information will be presented for the student to read and assimilate.

If the presentation uses a simulation to develop the material, you must be certain the simulation is a reasonably good model of the real-life situation it represents. It may be helpful to review conventional methods used to present the material, such as a laboratory activity, workbook exercise, or lecture presentation, and to adapt the approach you select to a computer based lesson. Further, you should also consider possible unconventional methods particularly suited to computer based instruction, such as a program in which a student creates a glossary of key words and terms for the subject he or she is studying, that will make the computer presentation distinctly unique and a synergistic complement to other, more traditional, teaching strategies. The great promise of computer based learning is the opportunity computers offer for presenting certain kinds of material in creative and imaginative ways not possible with traditional teaching techniques.

You may wish to have your educational computer program stand alone in presenting the material, or you may develop it as a supplement to other learning activities. For example, a drill and practice program may be developed to review the vocabulary words at the end of a chapter in a social studies textbook. Graphing data explained in a mathematical textbook may be supplemented by a program in which students input data and construct a graph of the data. A program may be developed for a science class in which data obtained in a student chemistry experiment is entered and used to calculate answers to problems.

"How well?", the fourth aspect of the teaching model, refers to the level of student achievement. This means you must decide how much of the material presented should be retained by the student or what level of skill the student should develop. At this point you may find it useful to write objectives that state precisely what should be the learning outcomes of the program. As explained in Section 1.5, learning objectives provide guidance in constructing the features of the program that present and develop the subject. They are also used to evaluate student achievement. For instance, you may wish to include a pretest based on the performance objectives that will be presented early in the program to assess the student's prior knowledge of the material.

Generally, the more time spent in planning a program before it is written, the less time it will take to write the lines of code and debug the program. At all costs avoid sitting down at the computer keyboard and composing your program directly on the video screen. To do so could quickly lead to frustration and much wasted effort. Begin with a lined writing tablet and pen or pencil. Concentrate on

the main idea around which the program will be developed. In many educational programs the learning centers on a drill and practice exercise such as spelling words correctly, changing words to the possessive case, or adding fractions. In some cases the heart of the program may be an algorithm for converting Fahrenheit to Celsius temperatures, or it may be a simulation that models a real-world phenomenon: for example, an ecological simulation that portrays the changing size of a population of rats living within a city block.

After you have the main idea of your proposed program in focus, move on to decide how the material will be presented to the student. Start writing lines of BASIC to program the main idea in its simplest form. Here are some examples of how to begin. In a Fahrenheit/Celsius temperature conversion program the programmer might start by writing lines of code that call for (1) the input of a Fahrenheit temperature, (2) the formula that converts the Fahrenheit input to a Celsius temperature, and (3) printing the equivalent Celsius temperature on the screen. In a drill and practice program for teaching the possessive case of nouns, the programmer might begin with lines that print each noun together with possible word endings for the possessive form. The student would be prompted to choose and input the word ending for the correct possessive form of each noun. For a program based on the ecology of a rat population, the programmer might use keyboard characters to plot a city block on the screen and then plot X's within the block to represent rats, each X representing 10 rats. Factors such as food sources, poisons, deaths, and births would be introduced as variables that alter the location and number of rats as the program runs.

At this point you would do well to remind yourself of the importance of interaction between the computer and student. If in your programming you find yourself writing lots of PRINT statements that display lengthy lines of text on the screen, you are definitely not going in the right direction. Stop! Reconsider what you are about. However, if at this point you are writing prompts and input statements that call for student responses, then your programming is moving down a productive path.

Once you have written the lines of code for the main part of the program and debugged any errors, you should move on to other pertinent considerations. Decide how long the program should be. Avoid making the running time too short, thus leaving the student unfulfilled or dragging out the material so that the student becomes bored. Think about it. How many temperature conversions must a student do to learn the conversion operation? How many possessive cases must a student identify correctly to get the point?

How long should the student manipulate the environmental factors affecting the rat population in order to learn the ecological principles involved? These questions are not always easy to answer precisely. Most often the decision will be based on experience and intuition.

Concerning the number of questions or problems to be used in a program, there does seem to be a connection between 5s and 10s (fingers and toes) and how many of these items are included. Experience indicates that problems or questions are best presented in sets of 5s or 10s with a maximum number within a program of 20 or 25. You may find it best to arrange each set of questions or problems in order of increasing level of difficulty—the easiest first and the most difficult last—unless there is some good reason for doing it some other way. If many questions or problems are needed they can be stored in data statements and accessed through random selection.

Judging the optimum length of a simulation program may be difficult. For the rat simulation the program should run long enough for several cycles of increasing and decreasing numbers of rats so that the student sees the cause and effect relationships involved. Initially, make your best judgment about the proper length of a program. You can modify it later, based on feedback from trials in which students use the program.

After you have the educational content programmed, add enhancements to increase the educational value and user friendliness of the program. For example, program positive reinforcements such as RIGHT ANSWER! and negative reinforcements such as SORRY, YOU MISSED IT to signal whether student responses are correct. Add counters that keep track of the numbers of correct and incorrect responses, and have the program print out the scores at the end.

Write the program so that the display on the video screen will have a positive visual impact. Above all, avoid having the screen look like a page out of a textbook. Unless there is some good reason for it, do not have the printing of text begin in the upper left corner, and absolutely avoid any scrolling of material up and off the top of the screen as the program executes. One useful guideline in formatting the video display is to print no more than six lines of text on the screen at one time. Center and space the material by PRINT and TAB statements. Generally, leave at least one blank line between successive lines of text. An exception is the display of a table or graph requiring the full screen. You will have to decide the exceptions to the rule. However, the less screen clutter, the easier it will be for the student to understand and learn from your program.

As your program nears finished form, give attention to the addition of input checks. At this point in the creation process you are

probably the only person running the program, and you know the kinds of inputs needed for it to run without errors. But it will be different with student users. For example, errors such as inadvertently hitting the Return key before typing an input or indiscriminately typing a word when a number is required are to be expected. Such input errors, if not checked and corrected, will upset the best of educational programs.

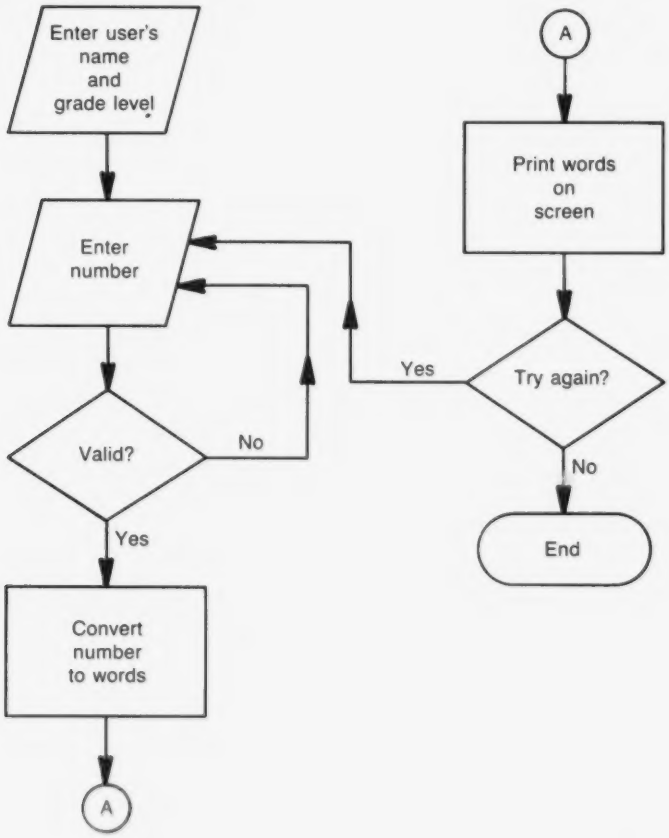


Figure 1.1
Outline of a drill and practice program. See Appendix A for a listing of The Number Machine.

Sketching an outline at some point in program development may help you visualize more concretely the program components and aid in understanding how the parts of the program interact. Examples of outlines for three kinds of educational computer programs are charted in Figures 1.1–1.3. These charts outline the modules that make the three programs and trace the flow of each. A listing for each program is in the Appendixes.

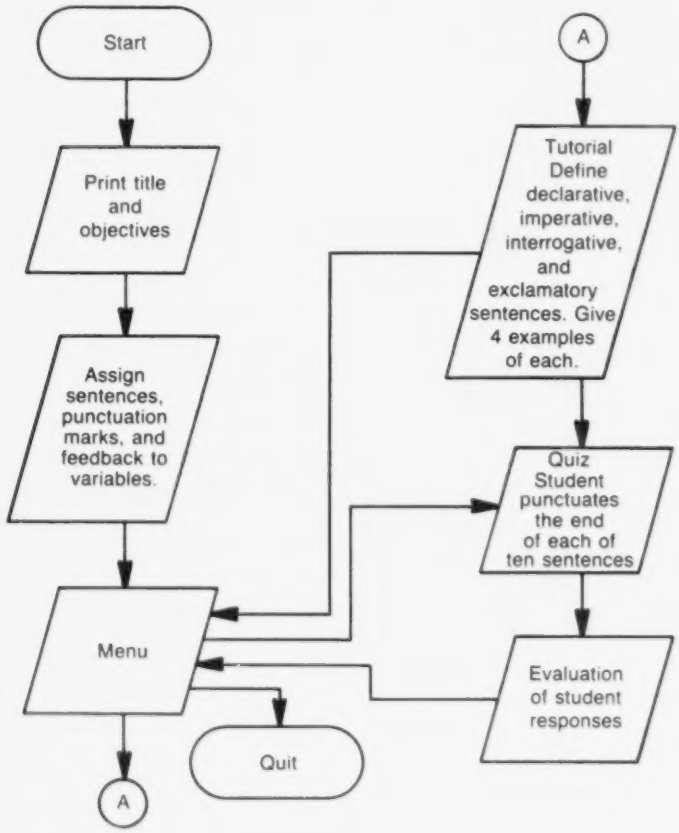

Start
Print title and objectives
Assign sentences, punctuation marks, and feedback to variables.
Menu
A
Quit
A
Tutorial Define declarative, imperative, interrogative, and exclamatory sentences. Give 4 examples of each.
Quiz Student punctuates the end of each of ten sentences
Evaluation of student responses

Figure 1.2
Outline of End Punctuation, a tutorial program listed in Appendix C.

Figure 1.3
Outline of Animal Heartbeats, a simulation program. A listing of this program is in Appendix D.

Writing an outline of a program will be easier if you compose the program in a structured format in which the different routines composing the program are readily identified in the listing (see Section 1.7). If you give some thought initially to writing your program as a series of routines and subroutines, you will find it easier to follow the program and modify it as you develop it into its final form.

Most programmers begin by typing line numbers in a sequence of 10's: 10, 20, 30, 40, 50, and so on. This is fine and works well until you begin to make changes and add lines until there are no more line numbers available for adding lines between those already written. This frustrating problem can be eliminated by using a line renumbering program that renumbers all lines in sequence by 10's, thereby making room for the lines to be added. In the renumbering process all GOTOs and GOSUBs are renumbered correctly to correspond with the new sequence of line numbers. A renumbering utility program is available on the Apple DOS 3.3 system master diskette.

After your program is completed and running satisfactorily, try it out with a few students. Let each student run the program. Take notes on difficulties each student encounters and on any program bugs. Do not help the student as he or she moves through the frames unless such help is absolutely necessary for the student to complete the program. After each student finishes, run through the program with the student and ask for comments and suggestions. Then revise the program based on this feedback.

In summary, be reminded that writing an educational program is a communications task. To be successful, your program must communicate the program's content effectively so that the student will understand and learn. Moreover, it is important to maintain a degree of flexibility throughout the design and development process. Some good ideas may come while you are programming at the keyboard or while jogging, shaving, bicycling, or hot tubbing. Do not hesitate to incorporate worthwhile changes, even though they involve additional programming time, that will increase the teaching effectiveness of your program. The resulting upscaled quality will well reward your extra efforts. In the following sections specific and concise directions are given for writing the routines that make up an educational computer program. Main program routines and subroutines that enhance the attractiveness, interactive features, and user friendly aspects of a program are discussed. These routines, together with some imagination as to the style and content of the program, will insure creation of a product that is functional and successful in teaching the topic presented.

ON LINE 1.4

1. State the four questions that conceptualize the teaching model discussed in this section.
2. Explain the purpose of each question in constructing an educational computer program.
3. Select a topic for an educational computer program and identify the main idea that will be the heart of the program.
4. Write an outline of the program selected in answer to On Line 3 that identifies the main parts of the program.
5. How can a chart be helpful in writing a program?
6. Explain the flow of the program charted in Figure 1.1.
7. Compare the charts for the programs of Figures 1.1–1.3. How are the programs alike? How are they different?

1.5 Learning Objectives

One feature you might like to include in your program is a concise statement of one or more learning objectives to be achieved by the student. A learning objective tells the student what he or she should get out of the program. A clearly stated objective identifies the learning a student acquires by using a program. Use of a learning objective in Animal Heartbeats is shown in Figure 1.4.

```
330  REM  PRINT LEARNING OBJECTIVE
340  HOME : VTAB (10)
350  SPEED= 100
360  PRINT "OBJECTIVE: TO COMPARE THE NORMAL RATE"
370  PRINT
380  PRINT "          OF HEARTBEAT FOR DIFFERENT"
390  PRINT
400  PRINT "          ANIMALS."
410  SPEED= 255
```

Figure 1.4
Line 350 in the program reduces the speed at which the objective is printed on the screen.

A learning objective stated in terms of performance contains a verb that expresses some type of observable action to be displayed by the student. Examples are "identify," "name," "list," "describe," "define," "solve," "compare," "contrast," and "predict." Each verb describes a specific behavior to be displayed by the student that can be observed by another individual. Stating an objective in terms of performance causes the objective to be definitive and precise, thereby making it comprehensible and useful.

Note that verbs such as "comprehend," "understand," and "know" are not appropriate in the statement of performance objectives, for these verbs do not designate behaviors to be displayed by the student. Statements such as "the learner will comprehend addition" or "the learner will understand the parts of speech in English" convey absolutely no information about specific behaviors to be exhibited.

Performance based learning objectives can be separated into three distinct parts: (1) description of the conditions under which the desired behavior will be observed; (2) description of exactly what behavior will be displayed; (3) specification of a level of acceptable performance. Here is a specific example:

```
10  PRINT "GIVEN A PROGRAM WRITTEN IN BASIC THE STUDENT WILL BE ABLE TO
    IDENTIFY AND CORRECT ALL SYNTAX ERRORS WITH 90 % ACCURACY."
```

Often a statement of the third part of a performance objective is omitted, and a level of 100 percent is assumed. Also part (1) can be omitted if the "givens" are obvious. Here are some examples of performance objectives stated in simple but concise terms. The student will be able to

```
10  PRINT "NAME THE CAPITAL OF EACH STATE IN THE UNITED STATES."
20  PRINT "IDENTIFY THE PARTS OF SPEECH IN A SENTENCE."
30  PRINT "ADD ONE-DIGIT NUMBERS."
40  PRINT "TYPE THE CORRECT PUNCTUATION AT THE END OF A SENTENCE."
50  PRINT "CLASSIFY CHANGES AS EITHER PHYSICAL OR CHEMICAL."
60  PRINT "LIST FOODS WHICH ARE MEMBERS OF THE FOUR BASIC FOOD GROUPS."
```

The first example would be appropriate in a program designed to teach state capitals. The second could be used in a program that presents examples of the parts of speech and requires students to identify them. The third is appropriate for a drill and practice program in mathematics. The fourth is appropriate in a program designed

to teach the punctuation marks used at the end of a sentence. The next objective is relevant to a computer program in science dealing with physical and chemical changes. The last objective would be listed in a program that teaches the characteristics of the four basic food groups and lists various foods that are members of each group.

Learning objectives provide more information than the behaviors to be displayed by the student and the level of acceptable performance. A well-constructed objective also suggests to the programmer what text must necessarily be included in the program to help the student achieve the objective. For example, suppose the objective stated in Line 10 is included in an educational program on verbs. (The YOU listed in the objective is the student.)

```
10 PRINT "GIVEN AN ENGLISH SENTENCE YOU WILL BE ABLE TO IDENTIFY THE VE
   RBS WITH 90 % ACCURACY."
```

Inspection of the objective indicates that if the learner has never heard of verbs, then the program must define verbs and show how to use them in sentences. The program would then provide examples of sentences and require the student to identify the verbs. We will discuss later the programming procedures related to evaluating student responses in terms of the acceptable performance criteria. Now we are concerned only with writing objectives. In terms of the objective in Line 10, would the program also expect students to construct their own sentences in English using verbs correctly? Absolutely not. The objective states nothing about the behavior "construct." The student is expected only to "identify." The programmer may include other objectives in the same program, one of which could relate to constructing sentences.

Here is another example.

```
10 PRINT "YOU WILL BE ABLE TO CONVERT LINEAR MEASUREMENTS IN THE METRIC
   SYSTEM TO LINEAR MEASUREMENTS IN THE ENGLISH SYSTEM."
```

Inspection of this objective indicates that the program will present only information on metric and English linear measurements, such as meters, centimeters, millimeters, yards, feet, and inches. The program will provide no instruction on any metric or English units other than linear units.

Learning objectives can be written to elicit behaviors at various levels of cognition. Benjamin Bloom's *Taxonomy of Educational Objectives Handbook I : Cognitive Domain* (David McKay, New York, 1956) is a useful guide for writing objectives, the achievement of which will tap the higher levels of students' thinking. The six major

categories in the *Taxonomy* are knowledge, comprehension, application, analysis, synthesis, and evaluation. Listed below are seven objectives that illustrate the range of levels in the *Taxonomy*. These objectives would be included in a tutorial program to encourage the student to interact with the program at the higher levels.

```
10  PRINT "AFTER READING A SHORT STORY PRESENTED BY THE COMPUTER"
20  PRINT "YOU WILL BE ABLE TO:"
30  PRINT "IDENTIFY THE MAIN CHARACTER"
40  PRINT "DESCRIBE THE PLOT"
50  PRINT "COMPARE AND CONTRAST THE MAIN CHARACTERS"
60  PRINT "ANALYZE THE PLOT DEVELOPMENT IN TERMS OF"
70  PRINT "SETTING AND THE MOTIVATION OF THE MAIN CHARACTER"
```

Examine the following lines from a simple program designed to illustrate the use of performance objectives in educational programs. The actual program on nouns would be much longer and more comprehensive.

```
10  PRINT "PLEASE ENTER YOUR NAME. ";
20  INPUT F$
30  PRINT "HELLO ";F$;".";" AFTER COMPLETING THIS PROGRAM YOU WILL BE AB
    LE TO IDENTIFY THE NOUNS IN A SENTENCE."
35  PRINT
40  PRINT "A NOUN IS THE NAME OF A PERSON, PLACE OR THING. HERE IS A SEN
    TENCE CONTAINING NOUNS."
50  PRINT
60  PRINT "STEVE DROVE HIS CAR TO THE GARAGE."
70  PRINT "THE WORDS 'STEVE', 'CAR', AND 'GARAGE' ARE NOUNS."
75  PRINT
80  PRINT "STEVE IS THE NAME OF A PERSON."
90  PRINT  TAB( 5)"CAR IS THE NAME OF A THING."
100  PRINT  TAB( 10)"GARAGE IS THE NAME OF A PLACE."
105  PRINT
110  PRINT F$;", CAN YOU FIND THE NOUN IN THE NEXT SENTENCE?"
112  PRINT
115  PRINT "COMPUTERS ARE FUN TO RUN."
117  PRINT
120  PRINT "TYPE THE NOUN HERE. ";
130  INPUT N$
140  IF N$ = "COMPUTERS" THEN  PRINT "VERY GOOD. YOU FOUND IT.": END
150  PRINT "SORRY, THE NOUN IS, COMPUTERS.": END
```

When writing an educational program, strive to simply and concisely state the learning objectives that the student is to achieve by using the program. The objectives may take the form of one or two statements printed on the screen, following the program's title, that serve as an introduction to the program. The learning objectives

should focus the student's attention on the essence of the program, indicating precisely what the student should learn. Focusing attention on learning outcomes aids the student in understanding and interpreting the purpose of the program's content, thereby resulting in higher student achievement.

---

1 What is a learning objective?

2 How are objectives useful in writing an educational program?

3 How are objectives useful to a student?

4 Where and how should objectives be included in an educational computer program?

5 Why are verbs such as "know" and "understand" inappropriate in the statement of performance objectives?

6 Describe the three parts of a performance objective and state the purpose of each. When may it be appropriate to omit one or more of these parts?

7 List five examples of performance objectives.

8 Choose a topic appropriate for an educational computer program and write three performance objectives for it.

9 Select five topics appropriate for educational computer programming and write a performance objective for each. Briefly describe what you would include in each program to help students achieve the objective.

1.6 Criteria for Design and Development

For an educational computer program to effectively fulfill its purpose, it must be designed for use by individuals with little or no previous computer experience. Nothing can be assumed! When developing such a program, you must consider various criteria. Just as there are accepted pedagogical techniques, so too are there

educational standards that should be followed when developing computerized activities for the classroom.

When considering the development of an educational computer program, one must certainly prepare well-defined objectives, as explained in Section 1.5. These objectives serve as a guide in the development of the program and will be used to eventually determine its degree of effectiveness. To achieve the desired purposes and make appropriate use of the computer, your program should meet the following minimal requirements.

1. Error free. All educational programs should be tried out with one or more students. Any program bugs or errors must be corrected. Many of the routines in this book will help you prevent and correct programming errors.

2. Grammatically correct. Spelling errors and improper punctuation tend to stand out on the computer monitor. Be sure to check the spelling and grammar in your PRINT statements.

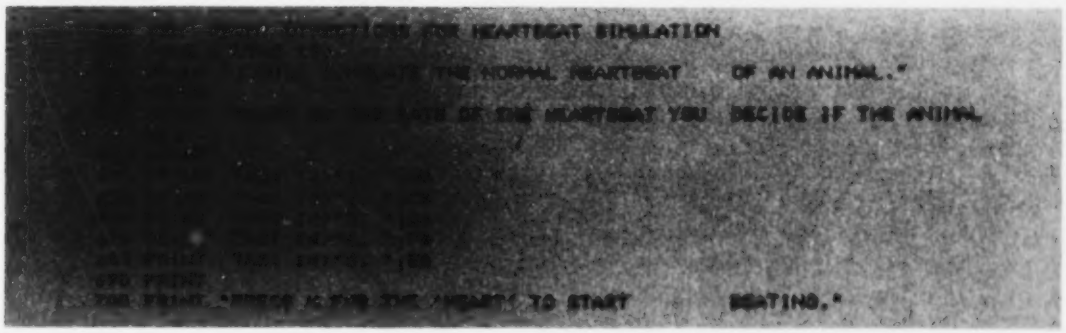
3. Factually accurate. The content of the program must be correct and factually accurate. This means errors in numbers, names, dates, and places must be avoided. The information must also be up to date. For example, a drill and practice program that teaches metric units should be based on SI, the modernized international system of metric units.

4. Compatibility level. The intended audience should dictate the degree of difficulty of the program. Vocabulary and concepts should correlate with the grade level of the user. Programs that are too elementary or too sophisticated will detract from the effectiveness of the lesson, allowing little learning to occur.

5. Motivating format. Just as classroom lessons need to be varied and interesting, so too must computerized educational activities. Self-pacing, immediate feedback, and other motivational factors inherent in computer based learning should be incorporated when plausible and possible.

6. Clarity of instruction. An educational program should contain precise directions that are sufficient to instruct the user about making selections and entering responses. These instructions should be given when necessary or when requested by the student, and repeated only when they are not followed. The instructions should leave no doubt about the procedures to follow.

Figure 1.5
Note the directions in Line 620 and 700.



7. Appropriate responses. When a student inputs information in response to questions or directions, the student's entry must be acknowledged and evaluated. If the entry is an acceptable or correct response, then positive feedback should be provided. Responses to incorrect or unacceptable entries should be corrective and free of harsh or demeaning content.

8. Acceptable language. As with any educational presentation, it is imperative to avoid derogatory remarks. All responses should be in good taste and free of slang, race, sex, slurs, and stereotypes. Never compromise good judgment.

9. Content continuity. An educational program should be designed so that it flows smoothly from one concept to the next. The presentation should be consistent throughout; that is, no discrepancies should be encountered in the various sections of the program. For instance, if UNITED STATES is an acceptable response in one part of a program, then THE UNITED STATES OF AMERICA should not be the required input for a similar question later in the same program. Maintaining specific and limited objectives is one way to insure a professional, accurate, and consistent presentation.

Most, if not all, of the foregoing requirements are second nature to users and creators of instructional material. It is beneficial, however, to review the criteria occasionally and be mindful of their importance in educational programs. Their application can contribute materially to the development of stimulating, worthwhile programs.

1 Why is it important to follow specified criteria when designing and developing an educational computer program?

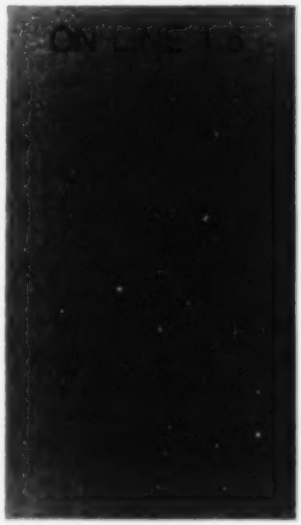
2 List the main design and development criteria discussed in this section.

3 How can the program developer gauge the level of difficulty for a program?

4 Discuss feedback and the handling of student inputs in an educational program.

5 What cautionary remarks are made in this section about "acceptable language?"

6 What is "content continuity," and what are its implications for educational programming?



Structured Programming 1.7

Have you ever had the frustrating experience of coming back after a period of time to a program you wrote in BASIC and finding parts of the listing difficult to read and comprehend? Quite likely you had to devote considerable time to rethinking the original assignment of variables and retracing the logic involved in the statements controlling the program run. This all too common experience can be alleviated by writing your BASIC programs in a structured format. Structured BASIC is the direct opposite of the "spaghetti" BASIC many beginners write in an introductory course. Because a structured BASIC program is infinitely easier to read and understand than an unstructured program, much less time is required to make revisions and add enhancements to the program. Moreover, with a structured program less time is expended in locating and correcting those common and not so common bugs that always find their way into a program as it is written.

Five essential characteristics of a structured BASIC program are

1 "Top down" programming
2 Limited use of GOTOs

3 Limited use of several statements on a line
4 Modules
5 Subroutines

Top down refers to programming in which the lines in the program are executed in sequential order. In reading a listing of the program, you can follow the logic and execution of the statements from one line to the next. A structured program has few, if any, GOTO statements. GOTOs branch a program unconditionally from one line to another, and their excessive use creates a scramble of branches that produce the spaghetti effect. Therefore, we recommend you generally avoid GOTOs in your programs and thereby develop the top down structured style.

Use a GOTO statement only when there is no reasonable alternative. For example, see Line 160. Execution of the GOTO statement depends on the condition given in the IF-THEN statement at the beginning of the line. If the condition is met, execution of the GOTO statement branches the program to Line 200.

```
160  IF ANS = "VERB" THEN  PRINT "CORRECT": GOTO 200
```

Another aspect of structured programming is a limit on the number of statements in a line. If you use a single statement per line it is easier to edit and correct your program when you locate bugs in it. Note the multiple statements in Line 150.

```
150 VTAB 10: HTAB 13: PRINT "VERB OR NOUN?": PRINT : PRINT "RUN IS A --
    --.": INPUT ANS
```

Line 150 is correct and will execute properly. However, if an editing change needs to be made, it may be more time consuming to make the change with multiple statements on a line than when there is only one statement. Many multiple line statements in a program create a complexity that interferes with debugging and efficient editing and causes a listing of the program to be difficult to read.

Line 150 may be written as a series of lines, one statement per line, as shown in the following listing. Note the increased clarity and readability that result in rewriting this one line, and recognize that the improvement is multiplied many times over when you write a whole program in this format.

```
150  VTAB 10
160  HTAB 13
170  PRINT "VERB OR NOUN?"
175  PRINT
180  PRINT "RUN IS A ----."
190  INPUT AN$
```

You may wish to modify the one statement per line principle by including more than one statement on a line when the statements are closely related in logic and function. For instance, you may find it convenient to use multiple line statements containing HTAB, VTAB, INVERSE, and NORMAL. However, in so doing you will be giving up the editing ease that occurs with single line statements. The following lines show two justifiable deviations from the one statement per line principle. Line 200 sets the coordinates for the inverse printing of X. Line 210 turns the inverse printing mode on, prints X, and turns the inverse mode off.

```
200  VTAB 12: HTAB 20
210  INVERSE : PRINT "X": NORMAL
```

A properly structured BASIC program is organized in the form of modules. Each module is a portion or unit of the program containing one or more routines that carry out a specific task. For instance, an educational computer program may contain a module that tabulates and prints the score on a quiz or a module that prints a table of data. Additional examples of modules are as follows:

- Prints a screen border and title
- Creates a menu
- Concatenates words into a sentence
- Prints a character graphic

In many instances a FOR/NEXT loop such as the one shown below may be used to program a module. The loop, properly designed, provides an efficient, powerful routine for many programming tasks because it offers maximum output with the fewest lines.

```
500  REM  PRINTS NAMES OF STUDENTS IN COLUMNS
510  FOR I = 1 TO N
520  PRINT S$(I),
530  NEXT I
```

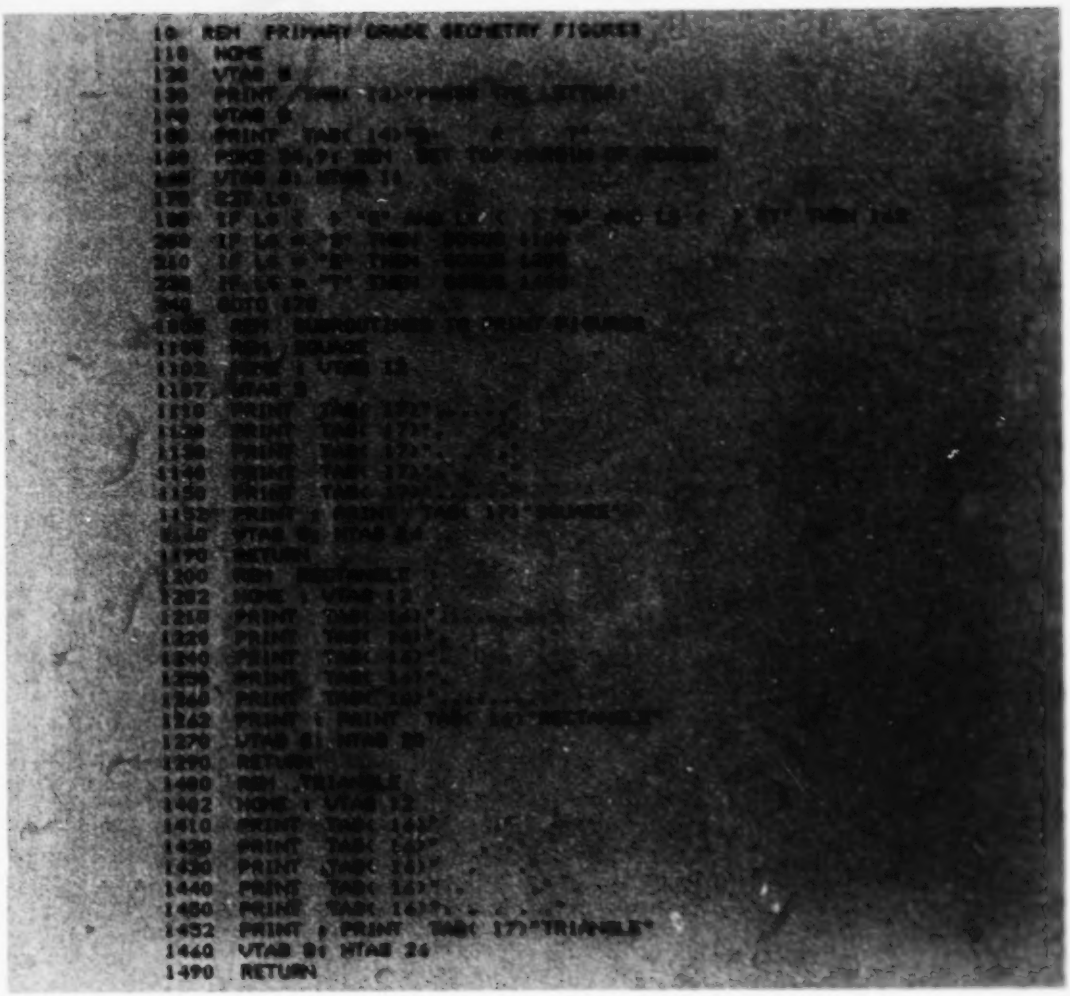
A FOR/NEXT loop contributes to the structuring of a BASIC program because there is an inherent logic associated with the lines in the loop and their execution. Statements within the loop are executed repetitively, the number of iterations being controlled and clearly defined by the FOR and NEXT statements. Moreover, the FOR and NEXT statements also denote the precise boundaries of the loop within a listing.

Remark (REM) statements, such as the one in Line 500, should be used to explain the functions of modules in your program. REM statements add clarity and understanding to programs whose listings might otherwise be difficult to follow. The remark in Line 500 helps a reader of the listing understand that the module prints the names of students assigned to the subscripted variable S$(I). The number of names printed is based on the value previously assigned to N. For example, if N = 10, the loop will run through 10 iterations and print 10 student names.

Use subroutines when writing a structured program. We recommend that you use a subroutine for each routine or module executed two or more times during a program run. Include a REM statement that explains what the subroutine does. Then use GOSUB statements in the main part of your program to call the subroutine for execution. An example of a structured program containing three subroutines is on page 29.

A final recommendation that will help immeasurably to make your program easier to read and comprehend, from the time you begin writing to weeks later when you are improving it, is to carefully select and define all variables. Nothing is more confusing than trying to follow improperly defined variables through an unstructured program. You can alleviate this problem by using mnemonics when selecting variables; in other words, choose letters for variables that will help you remember the words or numbers they represent. The variable AN$ used for storing an answer to a question is a good example of this principle. It contains the first two letters of the word "answer." Other examples of mnemonic variables include N for number, N$ for name, S$ for sentence, and CS$ for concatenated sentence. Also, list all variables used in your program, with their assignments, in REM statements at the beginning of the program. This way you can always refer to the REMs if you forget (and occasionally you will forget) the assignment of a variable.

Although individuals develop different personal styles in composing their programs, adherence to the following guidelines for writing structured BASIC will give your programs a consistent logic and clarity, make them easier to read, and greatly simplify the debugging process naturally inherent in all computer programming.



```
10  REM  PRIMARY GRADE GEOMETRY FIGURES
110  HOME
120  VTAB 8
130  PRINT TAB( 14)"MENU"
140  VTAB 9
150  PRINT  TAB( 14)"1. SQUARE"
160  POKE 34,0: REM  SET TOP WINDOW TO 0
165  VTAB 21: HTAB 14
170  GET L$
180  IF L$ < "1" AND L$ > "3" THEN 165
200  IF L$ = "1" THEN GOSUB 1100
210  IF L$ = "2" THEN GOSUB 1200
220  IF L$ = "3" THEN GOSUB 1400
240  GOTO 120
1000  REM  SUBROUTINES TO PRINT FIGURES
1100  REM  SQUARE
1102  HOME : VTAB 12
1107  HTAB 5
1110  PRINT  TAB( 17)"+---+"
1120  PRINT  TAB( 17)"|   |"
1130  PRINT  TAB( 17)"|   |"
1140  PRINT  TAB( 17)"|   |"
1150  PRINT  TAB( 17)"+---+"
1152  PRINT : PRINT  TAB( 17)"SQUARE"
1160  VTAB 21: HTAB 24
1170  RETURN
1200  REM  RECTANGLE
1202  HOME : VTAB 12
1210  PRINT  TAB( 14)"+-------+"
1220  PRINT  TAB( 14)"|       |"
1230  PRINT  TAB( 14)"+-------+"
1250  PRINT  TAB( 14)"|       |"
1260  PRINT  TAB( 14)"+-------+"
1262  PRINT : PRINT  TAB( 14)"RECTANGLE"
1270  VTAB 21: HTAB 24
1290  RETURN
1400  REM  TRIANGLE
1402  HOME : VTAB 12
1410  PRINT  TAB( 16)"/\"
1420  PRINT  TAB( 15)"/  \"
1430  PRINT  TAB( 14)"/    \"
1440  PRINT  TAB( 13)"/      \"
1450  PRINT  TAB( 12)"+--------+"
1452  PRINT : PRINT  TAB( 17)"TRIANGLE"
1460  VTAB 21: HTAB 24
1490  RETURN
```

Figure 1.6
Each of the three subroutines in this program is a module that prints a geometric figure.

Structured Programming Guidelines

1 Write the program so that the lines are executed sequentially, top down.
2 Use a GOTO statement only if there is no reasonable alternative.
3 Generally, limit each line to a single statement; use few multiple statement lines.
4 Write the program in a modular format.
5 Use a GOSUB for a routine that will be run two or more times.
6 Include a REM statement to explain each module. Include additional REMarks for complex lines or routines requiring clarification.
7 Define all variables in REM statements at the beginning of the program.

---

1 What are the characteristics of a structured BASIC program?

2 Why take the time to structure a program?

3 Contrast the top down style of programming with spaghetti programming.

4 Write a BASIC line that uses a GOTO and an IF-THEN conditional statement to send the program to a line that clears the screen.

5 Why is it recommended that you limit the number of multiple statement lines?

6 Write a program line that is an appropriate use of multiple statements.

7 How are mnemonics related to the selection of variables?

8 Explain the use of FOR/NEXT loops, subscripted variables, and subroutines in developing program modules.

9 Write a short program that contains two modules: one a FOR/NEXT loop, the other a subroutine.

10 Write a short structured BASIC program that calculates the area of a rectangle. The program should use a subroutine to print the reinforcement GOOD WORK, YOU GOT IT RIGHT when the student enters the correct answer. Be sure to include REM statements, and list all variables in one or more REM statements.

Programming the Content 1.8

The idea is born as a clear picture of your educational program emerges. Your screen blueprint is formulated, and the final phase of your endeavor is about to begin. It is now time to write the actual program. With anticipated frames in mind and learning objectives determined, you embark on the challenge of coding your program.

One good approach to the actual composition of an educational program consists of four steps:

1 Identify and initialize variables.
2 Develop logic flow.
3 Program subroutines.
4 Code the main program.

Adherence to these four steps will enable you to develop a precise and concise structured program that is easy to modify or enhance.

For illustration, let us assume that we are going to create a program to describe various four-sided geometric figures, and we will request the user to identify special characteristics of these shapes. After we have thought through the general design and objectives of this program, we should identify the required data and note the method by which this data will be introduced into the program. Then we construct a table, as shown in Table 1.1, composed of variables, data, and means by which the data will be assigned to their respective variables. As the program is created and refined, the table may be expanded and referenced as needed. It is important to keep the information organized, current, and readily available as the program is written.

A glance at the initial list reveals much about the yet-to-be-developed geometric shape identification program. For instance, we plan on asking the user's name and assigning it to N$. Continued

examination of the table reveals other insights, such as arrays to number the examples and name the shapes, and calculating the area of each shape. Writing the information in tabular form requires us to organize our thoughts as we begin the process of actually coding the program.

Table 1.1
Variables, data, and assignment methods for a geometry program

Variable	*Data*	*Assignment Method*
N$	Student name	INPUT
E( )	Example number	FOR/NEXT loop
A	Area	Calculated
FS( )	Name of figure (square, rectangle, parallelogram)	READ/DATA

When the variables have been assigned and their method of incorporation into the program has been noted, the logic flow of the program can be developed. An increasingly popular method of program development is to list appropriate REMarks that outline the flow and structure of the program. Using the sample data presented above, we develop the program as follows:

```
10   REM DESCRIPTION OF PROGRAM
20   REM  SHAPE IDENTIFICATION


100   REM IDENTIFY AND INITIALIZE VARIABLES
110   REM  N$ = STUDENT NAME


500   REM MAIN PROGRAM


2000   REM SUBROUTINES

10000   END
```

Having formulated the basic structure of the program, we now focus our attention on determining the contents of the subroutines. Subroutines must be written that create the four-sided geometric

shapes: square, rectangle, and parallelogram. With this in mind, our subroutine remarks may be expanded to include this information.

```
2000  REM SUBROUTINES

2100  REM SQUARE

2200  REM RECTANGLE

2300  REM PARALLELOGRAM
```

Once these subroutines have been determined, the main program can be formulated. The main program, in this example, will consist of a series of screen displays presenting information such as the name and area of the displayed geometric figure. Hence, the program remarks can be expanded to indicate these facts.

```
500  REM MAIN PROGRAM

600  REM SELECT NAME OF FIGURE

700  REM SELECT SUBROUTINE TO CREATE FIGURE

800  REM CALCULATE AREA
```

The general program outline is now taking shape. A data list has been made, and the flow of the program has been refined with the inclusion of specific subroutine remarks. The intent of the program has also been clarified by the remarks in the main program. Before focusing attention on the programming techniques used to construct and present the geometric figures called for, we give an abbreviated listing of the outline formulated thus far. It shows the emergence of a structured program, discussed in Section 1.7, with a specific purpose.

describe program
identify data
main program
    name of geometric figure
    display of figure
    calculation of area

```
subroutines
    construct square
    construct rectangle
    construct parallelogram
end
```

Every program is unique in that it requires the specific contents to be coded in a manner that corresponds to the educational objectives of the program. Our example necessitates the programming of routines to create various geometric figures that can be identified by a student. For example, to depict a rectangle, one need simply enter the program lines:

```
2210   PRINT "*****"
2220   PRINT "*****"
2230   PRINT "*****"
2240   PRINT "*****"
```

Note that the figure produced by a run of these lines will be a rectangle with dimensions 5 units wide by 4 units high. Now we must consider the printing of rectangles of various dimensions. The figure in the preceding listing has fixed sides of 5 units and 4 units, resulting in an area 20 square units. A rectangle 8 units by 3 units would require line statements such as 2210–2240, with eight asterisks in each of three PRINT statements. Similarly, a rectangle 10 units by 6 units would require 10 asterisks in each of six PRINT statements. The program could be written this way with a separate routine for each size of rectangle.

However, there is a simpler approach to the creation and display of variously sized rectangles. Though the algorithm needed to print almost any size of rectangle is more difficult to program than the simplistic PRINT statements, we can use these PRINT statements as a guide to create such a routine.

Because the sizes of the rectangles vary, variables must be assigned for the width and height of the figure. Letting W represent the width (we must remember to add it to the data table), we create a loop that will print W asterisks on a line.

```
2230   FOR J = 1 TO W
2240   PRINT "*";
2250   NEXT J
```

Letting H represent the height of the figure, we use a loop that executes the preceding FOR/NEXT loop H times to create the rectangle.

```
2220  FOR I = 1 TO H
2230  FOR J = 1 TO W
2240  PRINT "*";
2250  NEXT J
2260  NEXT I
```

With the exception of one program line, our rectangle is now complete. A PRINT statement must be inserted between the NEXT J and NEXT I (Lines 2250 and 2260) to act as a carriage return so that W asterisks are printed on H successive screen lines. A comparison of the initial and final routines reflects the degree of flexibility contained in the improved code.

```
2210  PRINT "*****"
2220  PRINT "*****"
2230  PRINT "*****"
2240  PRINT "*****"
```

```
2220  FOR I = 1 TO H
2230  FOR J = 1 TO W
2240  PRINT "*";
2250  NEXT J
2255  PRINT
2260  NEXT I
```

Our example illlustrates an important programming technique that is often overlooked. Many programmers attempt to create algorithms by plunging into complicated routines that involve nested loops and complex arrays. A simple listing of PRINT statements will usually reveal a pattern that can be transformed into a loop that expands the procedure to produce similar results.

The rectangle subroutine must also specify the location on the screen where the rectangle is to be printed. Using the TAB function, we can alter the nested loops (Lines 2220–2260) by inserting a program line such as

```
2225  PRINT  TAB( M);
```

A variable (M) is used so that placement can be controlled. This control, along with the selections of W and H (the width and height of the figure), originates in the main program. To determine an appropriate value of M, a value that will result in the centering of the rectangle on the screen, one need simply subtract the width of the figure (W) from the screen width of 40 and divide by 2. Thus the program line

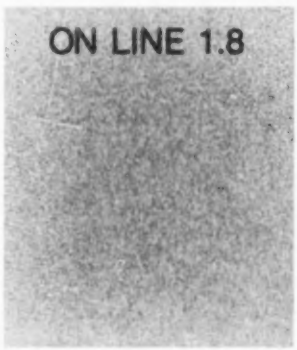
```
670  LET M = (40 - W) / 2
```

will be the appropriate TAB value to center the horizontal placement of the rectangle. An updated listing of the program reveals

a structure that is well on its way to becoming a creditable final product.

```
10  REM DESCRIPTION OF PROGRAM
20  REM SHAPE IDENTIFICATION

100  REM IDENTIFY AND INITIALIZE VARIABLES
110  REM N$ = STUDENT NAME
120  REM W = WIDTH OF FIGURE
130  REM H = HEIGHT OF FIGURE
140  REM M = CENTERING TAB

300  LET W = 0
310  LET H = 0
320  LET M = 0
500  REM MAIN PROGRAM

600  REM SELECT NAME OF FIGURE

640  REM SET WIDTH AND HEIGHT OF FIGURE

660  REM SET SCREEN PLACEMENT
670  LET M = (40 - W) / 2
700  REM SELECT SUBROUTINE TO CREATE FIGURE

800  REM CALCULATE AREA

2000  REM SUBROUTINES

2100  REM SQUARE

2200  REM RECTANGLE
2220  FOR I = 1 TO H
2225  PRINT  TAB( M);
2230  FOR J = 1 TO W
2240  PRINT "*";
2250  NEXT J
2255  PRINT
2260  NEXT I
2270  RETURN
2300  REM PARALLELOGRAM

10000  END
```

Walking through the beginnings of a program such as this geometric shape identification example should provide some insight into the preparation necessary to transform your educational idea into a computer program. Though all programs are unique in their output, the content of most programs could and should be structured so that the ideas can be developed in an organized manner. The outline created simplifies the task of writing the program. One may continue creating subroutines, enhance the main program, or develop the introduction. Because educational programs are frequently altered to meet the needs of the teacher, student, or text, the technique of outlining the program in REMark form allows the programmer to adjust the content for use with a variety of audiences.



1 What are the four steps recommended for composing an educational computer program?

2 List the ways in which a variable may be assigned.

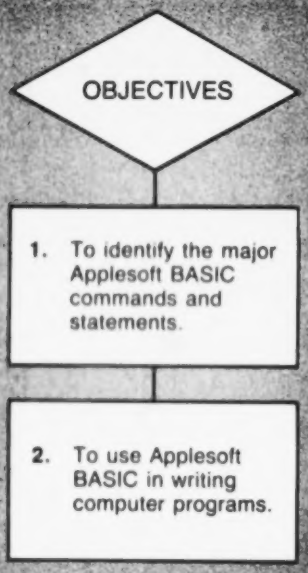
3 Construct a table that lists five variables, their assignments, and methods of assignment.

4 Why are REM statements used in a program?

5 Write an outline of a short program based on the techniques recommended in this section.

CHAPTER TWO

A SHORT REVIEW OF APPLESOFT BASIC


OBJECTIVES
1. To identify the major Applesoft BASIC commands and statements.
2. To use Applesoft BASIC in writing computer programs.

Major features of Applesoft BASIC are covered in this chapter as a review for those familiar with this version of BASIC and to identify the distinguishing features of Applesoft for those having programming experience with other versions of BASIC. Further, the chapter is organized so that it may serve as a ready reference for commands, statements, and functions used in programming the routines listed and discussed in the chapters that follow.

2.1 Variables

VARIABLE NAMES

Variable names are used to identify memory cells in the computer in which either a number or string is stored. A variable used to store a number is called a numeric variable, and a variable used to store a string is called a string variable. Strings contain one or more letters, symbols, or numbers and are identified by their enclosure within quotation marks. Some examples of strings are

"TEACHER" "ROOM 235" "SEVEN > FIVE"

Applesoft variable names must begin with a letter of the alphabet and may contain additional letters and numbers. Here are some examples of variable names:

Numeric variables
A A1 SUM AVERAGE STUDENT

String variables
A$ A1$ SUM$ AVERAGE$ STUDENT$

The only difference between the name of a numeric variable and a string variable is that the string variable must have a $ as the last character.

Applesoft BASIC recognizes and identifies only the first two characters of a variable name. Thus, when selecting variable names, you must make certain that what you believe are different variables really are different to the computer. If they are not different, strange programming errors may occur, the source of which can be difficult to locate. Avoid variables having the same first two letters, such as

SCHOOL$ and SCHOLAR$. Both are recognized by the computer as being the same string variable SC$. It is recommended that you use simple variable names of one or two letters or a letter and a number.

- N Number
- C Counter
- T Total
- A1 Average
- N$ Name
- Q$ Question
- SC$ School
- S2$ Sentence

Also, it is important to identify in REMark statements the variables you use in a program. This technique makes debugging and modifying a program much easier.

REMark statements such as Line 100 document aspects of a program and are not executed when the program is run.

```
100  REM  N = NUMBER  N$ = "NAME"
```

ASSIGNMENT TO VARIABLES

Four statements for assigning numbers and strings to variables are LET, READ/DATA, INPUT, and GET. The LET statement is discussed in this section; the others are discussed in the following sections.

LET Statement

The LET statement can assign either a number or a string to a variable, depending on the variable type. In Applesoft BASIC the LET statement must be preceded by a line number.

```
10  LET N = 16     20  LET N$ = "NAME"
```

In Line 10 the number 16 is assigned to the numeric variable N. In Line 20 the string NAME is assigned to the string variable N$. Note that the string is enclosed in quotation marks.

Once a variable has been assigned a value, the variable may be referenced anywhere in the program by a statement containing the

variable name. For example, Line 120 will print PRINCIPAL, the string assigned to P$ in Line 100.

```
100 P$ = "PRINCIPAL"
120  PRINT P$
```

String variables cannot be used in calculations involving mathematical expressions. Consider the following programs:

```
10  LET N$ = "4"
20  PRINT 2 * N$
30  END
```

```
10  LET N = 4
20  PRINT 2 * N
30  END
```

The program on the left will not run, and the message ?TYPE MISMATCH ERROR IN 20 will be printed because N$ contains the string character "4" and not the number 4. In the program on the right, N is assigned the number 4 in Line 10. Only numeric variables and numbers can be used by the computer in mathematical calculations.

Here is another example of an assignment to a string variable.

```
10  LET S$ = "CALIFORNIA STATE UNIVERSITY, NORTHRIDGE"
```

The string in Line 10 will be stored in memory location S$ for future reference in the computer program. The length of this string is 39 characters, including the spaces. An Applesoft string can have from 0 to 255 characters. If you try to include more than 255 characters in a string, an error message will be printed.

A string containing no letters, numbers, spaces, or other characters is the null string. An example of the null string is

```
10  LET Y$ = ""
```

There is no space between the two quotation marks that enclose a null string.

The LET statement may be used to assign arithmetic expressions to variables.

```
10  LET N = 4 * 7 + 6.83
20  LET H =  SQR (A * A + B * B)
```

In Line 10 the product of 4 and 7 added to 6.83 is assigned to N. In Line 20 the square root of the quantity A times A plus B times B is assigned to H.

Although the word LET may be used in assigning values to variables, Applesoft BASIC does not require LET in the assignment statement. Both of the following statements assign the number 40 to the variable A.

```
10 A = 40     10  LET A = 40
```

It is important that you clearly understand the meaning of the = symbol in an assignment statement. This symbol should be interpreted not as "equal," as in mathematics, but as "assign" or "replace." A = 40 means replace the contents of the memory cell named on the left side of the = by the value of the expression on the right side. Or we can say assign the value of the expression on the right side of the = to the variable named on the left side. Correct understanding of = in assignment statements gives meaning to a statement like

```
10 C = C + 1
```

In mathematics the statement in Line 10 is meaningless. But Applesoft BASIC understands it clearly: C = C + 1 means replace the contents of the memory cell called C with the old contents of C plus 1. If C contains the number 5, the statement C = C + 1 will replace the 5 with a 6. Statements like this are often used as counters in educational programs to keep track of a student's number of correct and incorrect responses and to monitor the number of opportunities given the user to answer a question correctly.

READ/DATA 2.2

READ statements are used with DATA statements to assign numbers or strings to variables. For example:

```
10  READ A
20  PRINT A
30  DATA  2
40  END
```

The READ statement in Line 10 assigns the 2 in the DATA statement (Line 30) to the numeric variable A. Line 20 prints the value, showing that the number 2 has indeed been assigned to A. Used in conjunction with a DATA statement, the READ statement is actually an assignment statement similar to a LET statement. If there

is a READ statement in a program, there must also be a DATA statement, or an error message will be printed.

The foregoing program has an END statement, but Applesoft does not require it. The program will run correctly if Line 40 is deleted. An Applesoft program will run until no more lines remain and will end after the last line is executed. DATA statements may be placed anywhere in a BASIC program, even before the READ statement. DATA statements, however, are unexecutable statements and are recognized only by READ statements. The items listed in DATA statements are read from left to right across all DATA statements, beginning with the first one in the program and continuing through all successive DATA statements. Each item in a DATA statement is separated by a comma:

```
200  DATA  90,85,73,50,95
```

If the program attempts to read beyond the last data item in the program, the error message ?OUT OF DATA ERROR IN X, is printed, where X indicates the line number of the READ statement trying to read the nonexistent data item.

Items to be read from a DATA statement can be placed in one or more DATA statements. Remember, data items are read from left to right in succession across all DATA statements.

```
10  READ N
20  PRINT 2 * N
30  GOTO 10
40  DATA  2,4,6,8,10,12

]RUN
4
8
12
16
20
24
```

```
10  READ N
20  PRINT 2 * N
30  GOTO 10
40  DATA  2,4,6
50  DATA  8,10,12

]RUN
4
8
12
16
20
24
```

```
10  DATA  2,4,6
20  DATA  8,10,12
30  READ N
40  PRINT 2 * N
50  GOTO 30

]RUN
4
8
12
16
20
24
```

These three programs produce the same output. The run of each program will terminate with a printing of the error message indicating the program is out of data that it can read. Although DATA statements may be placed anywhere in a program, we recommend that you place them immediately after the appropriate READ statement. This helps keep track of which READ statements are reading from which DATA statements and aids in debugging programs.

READ statements may contain more than one variable:

```
10  READ X,Y
20  DATA  4,8
30  PRINT X * Y
```

In Line 10 the first item in the DATA statement is assigned to X, and the second data item to Y. If the DATA statement in Line 20 contained only one item, the program would abort, and the error message indicating out of data would be printed. There must be a data item in a DATA statement to be read each time a variable is assigned by a READ statement. Otherwise, the program will terminate, and an out of data message will be printed.

DATA statements may also be used to assign string variables:

```
10  READ C1$,C2$
20  PRINT C1$;" ";C2$
30  DATA  ENGLISH 101,ALGEBRA B
```

Note that the strings in a DATA statement need not be enclosed in quotation marks. However, as with numbers, the strings are separated by commas. What happens if the string itself contains a comma? The string must then be enclosed in quotation marks, as shown in Line 90. When this program is run, Line 80 prints PERIOD 4, ENGLISH.

```
70  READ PC$
80  PRINT PC$
90  DATA  "PERIOD 4, ENGLISH"
```

DATA statements may contain numeric and string data:

```
110  READ C$,R
120  DATA  ENGLISH,20
130  PRINT "YOUR CLASS IS ";C$;" AND YOUR ROOM NUMBER IS ";R
```

If you include both numeric and string data in a DATA statement, you must make certain there is a one-to-one correspondence between the position of the variable in the READ statement and the position of the data item in the DATA statement. Numbers must be assigned to numeric variables, and strings must be assigned to string variables. For example, if Line 110 in the previous program is changed to 110 READ R,C$, an error will result because the string ENGLISH cannot be assigned to the numeric variable R.

2.3 INPUT and GET Statements

INPUT Statement

A teacher often receives input from students. Whether or not this student participation is anticipated, it is vital because the student is actively involved in the learning process. Input is an integral component of any communication. BASIC has a statement that makes it available as a means of communicating with a computer through a BASIC program. An INPUT statement is used in a BASIC program to allow the user to enter data from the keyboard while the program is running. To signal the user that an INPUT statement has been reached, a ? is printed on the screen. The ? is a prompt indicating that the program has temporarily halted and is waiting for the user to type a response from the keyboard.

```
10  INPUT N
20  PRINT N
```

If you run this program, a ? will appear on the screen, and nothing more will happen until you enter a response and press the Return key. But exactly what kind of data is the program expecting? A number or a string? We can see by the program listing that a number is expected because N is a numeric variable. But the user of the program does not see the listing, only the results of the run. What the computer is expecting as a response is unknown to the user. So it is essential that every INPUT statement be preceded by a PRINT statement containing a prompt that informs the user of the expected input.

```
10  PRINT "PLEASE TYPE A NUMBER FROM 1 TO 10"
20  INPUT N
30  PRINT N
```

The output from this program printed by Line 30 is the number entered by the user showing that the number was, indeed, assigned to the variable N.

Strings may also be assigned to string variables using INPUT statements:

```
50  PRINT "PLEASE ENTER YOUR NAME "
60  INPUT N$
70  PRINT "HELLO ";N$
```

When this program is run, it prints a hello greeting to the name entered in Line 60. Suppose RICHARD is entered in response to the prompt in Line 50. Line 70 will print HELLO RICHARD because RICHARD is the string assigned to N$ by the INPUT statement in Line 60. Note the space after the O in "HELLO " and the closing quotation mark in Line 70. If the Space Bar on the keyboard had not been used to insert this space, RICHARD would be printed next to HELLO with no intervening space: HELLORICHARD. The space is part of the string "HELLO ". Although the semicolon in Line 70 is optional, it helps to make the line more readable.

In the previous program the user's input is printed on the screen directly below the prompt PLEASE ENTER YOUR NAME. If you want the input to be printed directly after the prompt, then write Line 50 as

```
50  PRINT "PLEASE ENTER YOUR NAME ";
```

The semicolon at the end of the PRINT statement signals the computer that no line feed or carriage return is to be issued before the INPUT statement is executed.

In Applesoft BASIC an input may be combined with a prompt in the same statement. For example:

```
100  INPUT "PLEASE ENTER YOUR TEACHER'S LAST NAME ";N$
```

The INPUT statement in Line 100 prints the string enclosed in the quotation marks, but it does not print a ? on the screen. The sentence printed on the screen is a prompt that indicates to the user that an input is expected. If a student using the program enters GONZALES, then N$ = "GONZALES". This form of the INPUT statement may be used when you do not want a ? printed on the screen.

Here is one last example of INPUT.

```
210  INPUT "ENTER YOUR NAME ";N$
230  PRINT N$
```

When the program is run, there will be a problem with the input in Line 210 if the entry contains a comma. For example, if the user enters DOE,JOHN. The error message ?EXTRA IGNORED will be

printed to inform the user that the input up to the comma was assigned to N$, and the remainder of the input was ignored (N$ = "DOE"). Precise use of prompts to inform the user as to what kind of input is acceptable helps alleviate problems associated with undesirable entries. In this case, the prompt should be reworded to read ENTER YOUR FIRST NAME.

GET Statement

The GET statement, like INPUT, accepts a user's entry from the keyboard. However, there are important differences between GET and INPUT. GET accepts only a single keystroke (character) from the keyboard and does not print this character on the screen. Moreover, GET does not print a ? on the screen and does not require the user to press Return after entering a response. The program resumes execution immediately after a key is pressed.

2.4 Branching

GOTO Statement

Applesoft BASIC programs consist of a series of numbered statements that are executed in numerical sequence. However, sometimes the sequential flow of execution must be altered. Altering the normal sequence, called branching, is a fundamental programming technique for the execution of repetitive tasks. One method for branching a program is a GOTO statement, which sends the program to a line specified in the statement. For example, the GOTO in Line 120 sends the program execution back to Line 100:

```
100  PRINT "SCHOOL";
120  GOTO 100
```

Line 100 prints SCHOOL on the screen. Each execution of Line 120 returns the program to Line 100, causing SCHOOL to be printed repeatedly until the program is terminated by the user.

Although this illustration serves well as an example of the effect of a GOTO on the run of a program, it does not illustrate the practical use of this statement. A more realistic use of a GOTO is as follows:

```
230 X = X + 1
240 Y = Y + 2
245  PRINT "ENTER THE SUM OF ";X;" + ";Y
250  INPUT N$
260  GOTO 230

]RUN
ENTER THE SUM OF 1 + 2
?3
ENTER THE SUM OF 2 + 4
?6
ENTER THE SUM OF 3 + 6
?9
ENTER THE SUM OF 4 + 8
?

BREAK IN 250
```

(user pressed Control-C and Return)

In Applesoft an unassigned numeric variable automatically has a value of 0. Therefore, the value assigned to X after the first execution of Line 230 is 1 (X = 0 + 1). The value 2 is assigned to Y after Line 240 is executed (Y = 0 + 2). Line 245 prints ENTER THE SUM OF 1 + 2. The answer entered by the user is assigned by Line 250 to N$. Line 260 sends the program back to Line 230 to repeat the sequence. However, in each subsequent printing of Line 245 the value of X is increased by 1 and the value of Y is increased by 2, thereby creating a new problem to be solved by the user each time Line 245 is executed.

GOTO enables a programmer to direct the flow of a program to a specified line. The GOTO statement, however, lacks any conditions or qualifications as to whether the program should go to the line specified. To set conditions for the branching of a program, we need an IF-THEN statement.

IF-THEN Statement

An IF-THEN statement directs the computer to either execute or ignore the instruction following the THEN, depending on the condition specified by the IF:

```
490  PRINT "WHO WAS PRESIDENT DURING THE CIVIL WAR?"
495  INPUT P$
500  IF P$ = "LINCOLN" THEN  PRINT "CORRECT!": GOTO 520
510  PRINT "SORRY, WRONG ANSWER."
520  PRINT "WHO WAS THE FIRST U.S. PRESIDENT?"
530  INPUT P$
```

The IF in Line 500 evaluates P$ to determine if it has been assigned LINCOLN. If the condition is true, the THEN portion of the statement is executed and CORRECT! is printed. If the condition in Line 500 is false, the program continues to Line 510 and SORRY, WRONG ANSWER. is printed. Note the GOTO in Line 500 to branch the program to Line 520 and thereby skip Line 510.

Line 500 contains two statements separated by a colon (:). If the condition set by an IF statement is true, then all statements in the line are executed. Such is the case with the GOTO statement in Line 500. For a more detailed discussion of multiple line statements in programming, consult Section 2.5.

Relational symbols, including <, >, =, and <>, may be used between the words IF and THEN in an IF-THEN statement.

```
IF X <> Y THEN
IF Z <  = Y+4 THEN
IF A = C+D THEN
```

Applesoft BASIC permits the use of the logical operators AND, OR, and NOT to form more complex conditionals.

```
IF X < B AND Y > 12 THEN
```

In this example the statement following the THEN would be executed only if both conditions are true; that is, X is less than 8 and Y is greater than 12.

In the next example the statement following the THEN would be executed if either of the conditions is true: A is less than Y, or Z is greater than N.

```
IF A < Y OR Z > N THEN
```

The following program uses an IF-THEN statement in Line 30 to determine if a number entered by the user is odd or even.

```
10  PRINT "ENTER A WHOLE NUMBER."
20  INPUT N
30  IF N / 2 =  INT (N / 2) THEN  PRINT "NUMBER IS EVEN": END
40  PRINT "NUMBER IS ODD"
```

The IF condition is read as follows: If the value of N divided by 2 is equal to the integer (INT) of the value of N divided by 2. N/2 is equal to INT(N/2) only when a number is even. For example, 6/2 = 3 and INT(6/2) = 3. But for an odd number, say 7, 7/2 = 3.5 and

INT(7/2) = 3. Thus, 7 does not satisfy the condition for an even number stipulated in Line 30.

The IF-THEN statement can be used to compare strings:

```
10  IF "CAT" > "CAP" THEN  PRINT "CAT COMES AFTER CAP"
```

Comparison of the strings CAT and CAP can be made by the computer. Applesoft does not count and then compare the number of characters in the two strings to determine if the condition is true or false. Words are compared, one letter at a time, in terms of position in the alphabet. The first comparison is, does C come after C? No. Then does A come after A? No. Finally, does T come after P? Yes, T comes after P in the alphabet. So Applesoft determines that CAT is greater than CAP, and the message CAT COMES AFTER CAP is printed. Another way of expressing this idea is that CAP comes before CAT in an alphabetical listing of these words.

Comparison of strings is based on a comparison of the ASCII (American Standard Code for Information Interchange) code values of letters. ASCII code values for uppercase letters are ordered from 65 for A to 90 for Z. So in a construction such as IF "A" < "B", the computer is actually comparing the numerical values of the letters.

GOSUB Statement

A GOSUB statement is somewhat similar to a GOTO in that it causes a program to branch to the line number specified by the GOSUB. GOSUB means to go to a subroutine that begins with the designated line number, execute the subroutine, and return to the next statement following the GOSUB statement. A subroutine is one or more lines, apart from the main body of a program, that performs a special task. For instance, a subroutine may be used to check the input of a zip code to make sure the entered number contains exactly five digits.

Subroutines are usually placed at the beginning or end of a program. They are used for tasks that must be done more than once during the run of a program. Such use for repetitive tasks decreases the number of program lines needed to execute the tasks and thereby increases the efficiency of the programming. The following program uses a GOSUB in Line 120 and a subroutine beginning at Line 2000.

```
100  PRINT "WHAT CITY IS THE CAPITAL OF CALIFORNIA?"
110  INPUT A$
```

```
120  IF A$ = "SACRAMENTO" THEN  GOSUB 2000: GOTO 140
130  PRINT "SORRY, WRONG ANSWER."
140  PRINT "WHAT CITY IS THE CAPITAL OF ILLINOIS?"
1190  END
2000 R =  INT (3 *  RND (1) + 1)
2010  IF R = 1 THEN  PRINT "CORRECT"
2020  IF R = 2 THEN  PRINT "YOU GOT IT RIGHT"
2030  IF R = 3 THEN  PRINT "RIGHT ON"
2040  RETURN
```

In this program the answer entered by a student in response to the question in Line 100 is evaluated by the IF condition in Line 120. If the student has entered the correct answer, SACRAMENTO, the GOSUB statement in Line 120 is executed and the program branches to the subroutine beginning with Line 2000. This subroutine prints one of three positive feedbacks to the student's answer, depending on the random number (1, 2, or 3) generated in Line 2000. The RETURN in Line 2040 sends the program back to the GOTO 140 statement in Line 120. The return from a subroutine is always to the next statement following the GOSUB statement that called the subroutine. GOTO 140 branches execution to Line 140, which prints the next question.

The lines that make up the subroutine need be included only once in the program, regardless of the number of questions and student inputs. Repeated execution of the subroutine is possible through inclusion of a GOSUB statement after each input. When programming GOSUBs, make sure that each subroutine ends with a RETURN so that the program returns to the next statement following the GOSUB. END in Line 1190 terminates the run of the program and prevents the program from running into and executing the subroutine at the conclusion of the program.

2.5 Multiple Statement Lines

In some cases multiple statement lines can improve the logic flow of a program. Multiple statement lines refer to the use of two or more statements in the same program line. For example:

```
100  PRINT R$: GOTO 120
```

A colon separates multiple statements on one program line. Rather than writing three PRINT lines to insert three blank lines after a phrase, one may opt instead to write the three PRINT statements in the same program line:

```
10  PRINT "TODAY IS"
20  PRINT : PRINT : PRINT
30  PRINT "FRIDAY!"
```

Consider the task of checking for a specific input and printing a clarifying message if the user enters anything but the requested information. A typical program code to accomplish this task is as follows:

```
2   INPUT "ENTER A NUMBER GREATER THAN 9 ";N
4   IF N > 9 THEN 10
6   PRINT "ENTRY WAS NOT GREATER THAN 9"
8   GOTO 2
10   PRINT "THANK YOU"
```

A multiple statement line in this instance can improve the logic flow of the routine and maintain its top down structure. The following routine, which incorporates the multiple statement line technique, eliminates the "jumping" from Line 4 to Line 10 or from Line 8 back to Line 2.

```
2   INPUT "ENTER A NUMBER GREATER THAN 9 ";N
4   IF N <  = 9 THEN  PRINT "ENTRY WAS NOT GREATER THAN 9": GOTO 2
6   PRINT "THANK YOU"
```

In the listing, program control remains at Lines 2 and 4 until a valid entry is entered. The multiple statements in Line 4 prevent program flow to any part of the program beyond the check.

When a GOSUB statement is used, program control returns to the statement immediately following the GOSUB statement after the subroutine is executed. The next statement may or may not be the next program line, depending on whether the line with the GOSUB is a multiple statement line. The following listing and run illustrates the order of statement execution when a GOSUB is contained in a multiple statement line.

```
 100   PRINT "YOUR GRADE IS ": GOSUB 500: PRINT G
 110   PRINT "CONGRATULATIONS"
 120   END
 490   REM SUBROUTINE
 500   LET G = 95
 510   RETURN

]RUN
```

```
YOUR GRADE IS
95
CONGRATULATIONS
```

In this program, execution begins at Line 100, where the phrase YOUR GRADE IS is printed on the screen. The GOSUB 500 statement then transfers program control to Line 500, where G is set to 95. The RETURN in Line 510 sends program control to the statement following GOSUB 500 in Line 100. Thus, the value of G, 95, is then printed on the screen before the subsequent program lines are executed.

Multiple statement lines are also frequently used with REMark statements. The next line illustrates the placement of a remark on the same program line as an assignment statement.

```
50  LET T = 0.06 * P: REM CALCULATE SALES TAX
```

Though multiple statement lines can improve the coding and efficiency of a program, they can also be the source of difficulty. One problem arising from their use is debugging. Consider the following two programs:

```
 10  LET AC = 4
 20  LET  AT  = 5
 30  PRINT AC +  AT
]RUN
?SYNTAX ERROR IN 20
```

```
 10  LET AC = 4: LET  AT  = 5: PRINT AC +  AT
]RUN
?SYNTAX ERROR IN 10
```

Since most Applesoft error messages note the line number in which an error occurs, the greater the number of statements per line, the more difficult it is to debug if an error occurs. Unless the overall structure, flow, or readability of a program is improved by using multiple statement lines, it is recommended that only one statement be written on a program line.

2.6 FOR/NEXT

A FOR/NEXT loop is a programming technique that can be employed when a set of instructions is to be executed a specific number of times. A FOR/NEXT loop provides an efficient method of coding repetitive tasks that otherwise can be tedious and time

consuming. For example, in a program to teach the vowels (A,E,I,O,U), a row of 40 A's can be printed with the line

```
5 PRINT "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

or with a FOR/NEXT loop:

```
10  FOR I = 1 TO 40
20  PRINT "A";
30  NEXT I
```

Although the FOR/NEXT loop requires three lines of code, at a glance it provides an accurate count of the number of times the statement in Line 20 will be executed. Line 10 initializes the counter variable (I) to 1. I, sometimes referred to as the index variable of the loop, will eventually be assigned all integer values from 1 to 40, inclusive. Thus Line 20 will be executed 40 times, resulting in a row of 40 A's.

The 1 in Line 10 is the initial value of I, and 40 is the terminal value. Line 20 is the execution statement of the loop. In this instance an A will be printed. The trailing semicolon is used to keep each A on the same screen line. Line 30 is the end of the FOR/NEXT loop. Execution of this line sends program control back to the corresponding FOR statement (Line 10) where the index variable is incremented by 1 and the statement within the loop is repeated.

One distinct advantage of using a FOR/NEXT loop to perform a series of repetitive instructions becomes obvious when the task of printing a single line of A's is changed to printing a screen of A's. With the FOR/NEXT loop coding technique, a simple change in Line 10 will do the job.

```
10  FOR I = 1 TO 800
20  PRINT "A";
30  NEXT I
```

Now 800 A's, or 20 rows of 40 A's, will be printed on the screen.

To emphasize the fact that the letter is a vowel, you can start each row of A's with the word VOWEL by using a nested loop as shown:

```
6  FOR I = 1 TO 20
8  PRINT "VOWEL ";
10  FOR J = 1 TO 34
20  PRINT "A";
30  NEXT J
40  NEXT I
```

The loop created by Lines 10–30 is referred to as the "inner" loop, and it will print a row of 34 A's as indicated by the terminal value of J in Line 10. However, this loop is contained, or nested, in another FOR/NEXT loop beginning at Line 6. This "outer" loop (Lines 6–40) will repeat 20 times, each time printing the word VOWEL before executing the inner loop and printing a row of 34 A's.

In educational programs FOR/NEXT loops are frequently used to create tables of information. The following loop creates a table of squares from 1 to 5.

```
10  PRINT "NUMBER","SQUARE"
20  FOR I = 1 TO 5
30  PRINT I,I * 2
40  NEXT I

]RUN
NUMBER          SQUARE
1               1
2               4
3               9
4               16
5               25
```

Note the use of the variable I in the loop. I is a mnemonic for increment, meaning small change. The incremental change from initial value to final value in the FOR statement is automatically set to 1. This value can be changed by adding the word STEP and a number to the FOR statement. The following program illustrates the placement of STEP in a FOR/NEXT loop and the resulting output.

```
10  PRINT "NUMBER","SQUARE"
20  FOR I = 1 TO 5 STEP 2
30  PRINT I,I * 2
40  NEXT I

]RUN
NUMBER          SQUARE
1               1
3               9
5               25
```

In this program STEP 2 increments the initial value of I two units each time through the loop. STEP can be followed by any numerical value, including decimals and negative values. Consider the following listings:

```
10  FOR I = 1 TO 3 STEP .5
20  PRINT I
```

```
10  FOR I = 4 TO 1 STEP  - 1
20  PRINT I
```

```
 30  NEXT I

]RUN
1
1.5
2
2.5
3
```

```
 30  NEXT I

]RUN
4
3
2
1
```

The absence of a statement in the body of a FOR/NEXT loop creates an interesting result. Enter and run the following program.

```
 10  FOR I = 1 TO 100: NEXT I

]RUN

]
```

Though no output is displayed on the screen, there is a pause before the BASIC prompt (]) reappears. Incrementing the final value of I will lengthen the pause or delay.

```
 10  FOR I = 1 TO 2500: NEXT I

]RUN

]
```

Note the length of time between the start of the run and the reappearance of the cursor on the screen. Since a FOR/NEXT loop with no statements in the body of the loop is used to create a pause in the run of a program, the variable P is often used as the index variable.

```
10  FOR P = 1 TO 2500: NEXT P: REM DELAY LOOP
```

Subscripted Variables 2.7

Often it is necessary to enter a set of numbers or strings into the computer's memory so that the program can operate on the set as a group. For example, if the computer is to sort a list of numbers from lowest to highest value, all the numbers are first entered into the computer's memory before the sort is accomplished, since the computer must compare each number with every other number. The same is true if the computer is required to alphabetize a list of names.

All the names are entered into the computer's memory so that each name may be compared with every other name.

BASIC permits us to deal with groups of related data as arrays. You can visualize an array as a group of contiguous boxes or cells in the computer's memory. Each box can hold a number or string and is referenced by an integer called an index or subscript. The subscript indicates to the computer which box is being referenced. An array of numbers is shown in Figure 2.1, where X(1), X(2), X(3), X(4), and X(5) are subscripted variables. The number in each set of parentheses is the subscript. In the figure are five boxes designated by the subscripts 1 to 5. Inside each box is a number. Think of the boxes as a group of memory cells in the computer, each of which contains a number. The subscripts indicate to the computer where each cell is located. The entire array (group of boxes) is given one variable name, X(y), where y is a positive integer or an expression that evaluates to an integer. X indicates that the variable is numeric and the parentheses following the X indicate that it is also a subscripted variable.

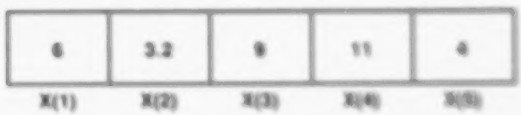


Figure 2.1
Array X.

Consider the following program, which prints the value contained in each of the cells.

```
 2  LET X(1) = 6
 3  LET X(2) = 3.2
 4  LET X(3) = 9
 6  LET X(4) = 11
 8  LET X(5) = 4
 10   PRINT X(1)
 20   PRINT X(2)
 30   PRINT X(3)
 40   PRINT X(4)
 50   PRINT X(5)


]RUN
 6
 3.2
 9
 11
 4
```

As the program run indicates, the numbers in the boxes designated by the respective subscripts are printed by Lines 10–50.

A more efficient procedure, using a FOR/NEXT loop, for printing the values stored in a subscripted variable is indicated next:

```
10  FOR I = 1 TO 5
20  PRINT X(I)
30  NEXT I
```

As the loop in the program iterates, I takes on successive values from 1 to 5, thus referencing each box in turn and printing the values contained in these boxes.

A FOR/NEXT loop may also be used to assign values to a subscripted variable. Consider array A in Figure 2.2. There are six boxes or cells in array A, referenced by the subscripts 1 through 6. Each box holds a number. To create the array the computer must assign the numbers to the subscripted variable that designates the array. This assignment may be made with a FOR/NEXT loop:

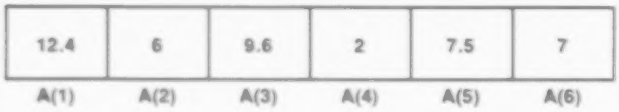


Figure 2.2
Array A

```
10  FOR I = 1 TO 6
20  READ A(I)
30  NEXT I
40  DATA  12.4,6,9.6,2,7.5,7
```

The numbers in the DATA statement (Line 40) are assigned to the subscripted variable A(I) by Line 20. As the loop iterates, the value of I increments from 1 to 6, thereby incrementing the subscript of A(I) from 1 to 6.

The routine may be changed so that the subscripted variable is assigned numbers entered on the keyboard by the computer user. In this case Line 20 is changed to an INPUT statement and Line 40 is deleted:

```
20  INPUT A(I)
```

When the numbers are assigned to the subscripted variable from the keyboard, the user must make six separate entries, one for each execution of the INPUT statement.

Applesoft BASIC automatically assigns 11 boxes or cells, numbered from 0 to 10, to a subscripted variable, whether they are

used or not. If you program a subscripted variable requiring more than 11 boxes, you must dimension the subscripted variable with a DIM statement:

```
10  DIM T(20)
20  FOR I = 1 TO 20
30  INPUT T(I)
40  NEXT I
```

The DIM statement in Line 10 causes the computer to allocate 21 memory locations (boxes) for the subscripted variable. Why 21 boxes? Applesoft always begins with a 0 when assigning boxes for the storage of values in an array. Thus, there will be 0 to 20 boxes allocated to the subscripted variable. Use of the 0 memory location, subscripted variable T(0), is optional with the programmer. Remember, however, that the number in the DIM statement sets the value of the highest subscript.

Subscripted variables can be used to assign strings. You need only include a $ in the variable name, designating it as a string variable. For example, A$( ). Up to 255 characters can be stored as a string in each cell of an array. In the following routine the names of four courses are stored in the array designated by T$(I). Note the structure of the array shown in Figure 2.3.

```
10  FOR I = 1 TO 4
20  READ T$(I)
30  DATA  ENGLISH 101,ALGEBRA B, ECONOMICS 400, HISTORY 250
40  NEXT I
```

Figure 2.3
An array with four strings assigned to a subscripted variable.

This section has been limited to a discussion of single subscripted variables and arrays having only one column. Such arrays are referred to as lists or vectors. Variables with more than one subscript can also be used in Applesoft programs. An example of a double

subscripted variable is A$(1,4). Programming multiple subscripted variables is beyond the scope of this chapter. Consult an Applesoft programming manual if you plan to use multiple subscripted variables in a program.

String Functions 2.8

When working with strings such as names of people or places, you may find it necessary to check one or more characters or spaces within the string. For example, if a student is asked to enter the name of the country on the northern border of the United States and enters an incorrect response, you may wish to determine if the entry's first letter is correct. To do this, you can use the LEFT$ function, a BASIC function that reads a specific number of characters starting with the leftmost character. To see how this function works, examine the following program lines and the resulting output.

```
 10  LET B$ = "CANADA"
 20  PRINT  LEFT$ (B$,1)

]RUN
C
```

The output generated by Line 20 is determined by the B$ inside the parentheses, which represents the string to be examined; 1 signifies the number of characters to assign to the LEFT$ function. Thus, C, the first character in CANADA, is printed by Line 20. If the first two characters are to be printed, then Line 20 becomes

```
20  PRINT  LEFT$ (B$,2)
```

and the resulting output would be CA.

Incorporating this string manipulation technique in a short routine can be accomplished as follows:

```
 10  LET B$ = "CANADA"
 20  INPUT "WHAT COUNTRY BORDERS THE U.S. ON THE NORTH?";AN$
 25  IF AN$ = B$ THEN  PRINT "CORRECT": GOTO 45
 30  IF  LEFT$ (AN$,2) =  LEFT$ (B$,2) THEN  PRINT "YOU'RE ON THE RIGHT T
     RACK": GOTO 45
 40  PRINT "SORRY"
 45  END
```

The LEFT$ in Line 30 is used to determine if the first two letters of the student response, assigned to AN$, match the first two letters of the correct answer assigned to B$. If they do, then the message YOU'RE ON THE RIGHT TRACK is displayed.

Similar to LEFT$ is the Applesoft function RIGHT$, which reads the rightmost characters of a string:

```
 10 D$ = "TUESDAY"
 20  PRINT  RIGHT$ (D$,3)

]RUN
DAY
```

As with LEFT$, the output generated by Line 20 is determined by a variable, D$, and a number, 3. D$ represents the string TUESDAY, and 3 signifies the number of characters assigned to the RIGHT$ function; thus, the three rightmost characters of D$ will be printed.

RIGHT$ can also be used to check a response in an educational program. The following listing illustrates the use of the RIGHT$ function to determine if an autumn month is entered.

```
10  INPUT "NAME AN AUTUMN MONTH ";AM$
20  IF AM$ = "SEPTEMBER" THEN 70
30  IF AM$ = "OCTOBER" THEN 70
40  IF AM$ = "NOVEMBER" THEN 70
50  IF AM$ = "DECEMBER" THEN 70
60  IF  RIGHT$ (AM$,3) = "BER" THEN  PRINT "INCORRECT SPELLING. TYPE THE
    WORD AGAIN AND PRESS RETURN. ": GOTO 10
65  PRINT "SORRY": END
70  PRINT "CORRECT"
```

If the student entry assigned to AM$ is not one of the fall months, as listed in Lines 20–50, but ends with the letters BER, then the RIGHT$ function in Line 60 will catch this response and the remark INCORRECT SPELLING. TYPE THE WORD AGAIN AND PRESS RETURN. will be printed.

The most powerful string function in terms of versatility is the MID$ function. The MID$ function can be used to read any consecutive sequence of characters. For example, the following program line will print the middle two characters in the word COMPUTER.

```
 10  PRINT  MID$ ("COMPUTER",4,2)

]RUN
PU
```

The number 4 immediately following the string COMPUTER signifies the position of the first character to be read; the number 2 determines the number of characters to be read. If you wanted to print the letters MPUTE from the string COMPUTER, change Line 10 to

```
10  PRINT  MID$ ("COMPUTER",3,5)
```

The following example illustrates a method by which substrings created by the MID$ function can be assigned to variables.

```
 10  LET W$ = "DEPARTMENT"
 20  LET P$ =  MID$ (W$,3,4)
 30  PRINT P$

]RUN
PART
```

Line 10 assigns the string DEPARTMENT to W$. The MID$ function in Line 20 reads four characters in W$S starting with the third character in the string. These four characters, PART, are then assigned to P$, which is printed on the screen.

The string functions LEFT$, RIGHT$, and MID$ are frequently used with variables in FOR/NEXT loops to produce a variety of text output. For instance, to illustrate the meaning of the word VERTICAL, you can use a FOR/NEXT loop and the MID$ function to print it vertically!

```
 10 V$ = "VERTICAL"
 20  FOR I = 1 TO 8
 30  PRINT  MID$ (V$,I,1)
 40  NEXT I

]RUN
V
E
R
T
I
C
A
L
```

Line 10 assigns VERTICAL to V$. The FOR statement in Line 20 generates eight iterations of the loop. In the first iteration, when the value of I is 1, Line 30 will print the first letter in VERTICAL.

The second time through the loop, when I = 2, Line 30 will be identical to

```
30  PRINT  MID$ ("VERTICAL",2,1)
```

meaning that the second letter in VERTICAL will be printed. Only one letter will be printed because the last number in the function, 1, denotes the number of letters to be read.

To illustrate a geometric figure, you can use the LEFT$ function with TAB and a FOR/NEXT loop. The following routine prints a triangle on the screen. In Line 20 the TAB function sets the screen location (column) for the printing of T each time Line 20 is executed.

```
10  FOR I = 1 TO 9 STEP 2
20  PRINT  TAB( (31 - I) / 2); LEFT$ ("TRIANGLES",I)
30  NEXT I

]RUN
               T
              TRI
             TRIAN
            TRIANGL
           TRIANGLES
```

Line 10 will set values of I to 1, 3, 5, 7, and 9. The first time through the loop the TAB will be set to (31 – 1)/2 or 15. Therefore, T is printed at Column 15. Since I = 1, the leftmost character of the word TRIANGLES, T, will be printed. The second iteration of the loop will produce a TAB of (31 – 3)/2 = 14, and since I is now 3, the first three letters of the word TRIANGLES will be printed at Column 14. The loop will continue until the output shown is produced.

2.9 Concatenation

Concatenation is the process of uniting two or more strings by a plus sign. For example, if B$ = "RAIL" and E$ = "ROAD", then B$ + E$ = "RAILROAD". This concatenation process can be coded as follows:

```
10  LET B$ = "RAIL"
20  LET E$ = "ROAD"
30  PRINT B$ + E$

]RUN
RAILROAD
```

Interchanging the string variables in Line 30 will rearrange the output:

```
 30  PRINT E$ + B$

]RUN
ROADRAIL
```

Concatenation is a string operator, meaning that strings must be used if the result is to be created by linking the terms together. Note the output of each listing below:

```
 10  PRINT 1 + 2

]RUN
3
```

```
 10  PRINT "1" + "2"

]RUN
12
```

The plus sign (+) in Line 10 adds the numbers when quotes are omitted. Converting to strings by inserting quotes results in the plus sign linking the digits to create 12.

Concatenation is frequently used in a BASIC program to combine first and last names, as illustrated in the following program:

```
 10  INPUT "FIRST NAME ?";F$
 20  INPUT "LAST NAME ?";L$
 30  LET N$ = F$ + " " + L$
 40  PRINT "HOW ARE YOU ";N$

]RUN
FIRST NAME ?JACK
LAST NAME ?SPRAT
HOW ARE YOU JACK SPRAT
```

Line 30 joins the first name entered with the last name entered. The quotes between F$ and L$ insert a space between the variable values; thus the value of N$ becomes JACK SPRAT and may be referenced at any point in the program.

CHR$ and ASC 2.10

Each character on the computer's keyboard is assigned a numerical value. These numerical values are standardized for most computers and are referred to as ASCII numbers. The ASCII code for the capital letter A, for example, is 65. Typing PRINT CHR$ (65) and pressing

Return will produce the output A. (Appendix F is a table of ASCII character codes.) CHR$ is a string function that returns the character assigned to the ASCII number referenced.

The CHR$ function is frequently used in BASIC programs to check for a specific keypress. For instance, if a screen prompt directs the viewer to press the Return key, then the program must be coded to check for such a keypress. The following example illustrates the use of the CHR$ function to determine if the Return key is pressed:

```
10  PRINT "PRESS RETURN TO CONTINUE ";
20  GET K$
30  IF K$ <  >  CHR$ (13) THEN 20
```

Since the ASCII code of the Return key is 13, Line 30 checks to determine if the key pressed has an ASCII value of 13.

The ASC function is the reciprocating function to CHR$; that is, it will determine the ASCII number associated with each keyboard character. For example, PRINT ASC ("B") will print the number 66, the ASCII value of "B". The ASC function can be used in lieu of CHR$ in many circumstances. The previous keypress routine can be altered to incorporate the ASC function instead of CHR$:

```
10  PRINT "PRESS RETURN TO CONTINUE ";
20  GET K$
30  IF  ASC (K$) <  > 13 THEN 20
```

2.11 LEN

The function LEN allows a programmer to determine the length of a string. LEN is a numerical value equal to the number of characters, including spaces, in the string.

```
 10  LET S$ = "INCH"
 20  PRINT  LEN (S$)

]RUN
4
```

Since the word INCH in Line 10 has four characters, its subsequent length is 4. The LEN function is assigned to a numeric variable for ease of reference later in a program.

```
 10  LET HH$ = "HUBERT H. HUMPHREY"
```

```
 20  LET H =  LEN (HH$)
 30  PRINT H

]RUN
18
```

Note the output 18. Even though the name assigned to HH$ contains 15 letters, all keyboard characters, including spaces, are counted. Thus the two spaces and the period are reflected in the total length of the string.

LEN can be used in an educational program to verify information. The following program asks the student to enter a telephone area code:

```
145 INPUT "ENTER AREA CODE ?";AC$
155 IF  LEN (AC$) <  > 3 THEN  PRINT "ENTER EXACTLY 3 DIGITS": GOTO 145
```

If the entry is less than three digits or greater than three digits, then an error message is printed and the user is asked to reenter the information.

VAL and STR$ 2.12

Applesoft BASIC includes several conversion functions, two of which are VAL and STR$. VAL converts any string variable to its numeric value. For example:

```
80 RM =  VAL (RM$)
```

If the string value of RM$ is "13", then the numeric value assigned to RM by Line 80 is 13. The following listing illustrates the placement of the VAL function in a program and the corresponding output:

```
 110  LET C$ = "8.20"
 120  PRINT "C$ = ";C$
 130  LET C =  VAL (C$)
 135  PRINT "C = ";C
 140  PRINT "C + 5 = ";C + 5

]RUN
C$ = 8.20
C = 8.2
C + 5 = 13.2
```

If a string is a group of digits, then its numerical value can be assigned to a numeric variable, as in Line 130. There is no trailing zero after conversion because numeric variables are assigned quantities and trailing zeros are irrelevant; hence they are ignored in printouts and calculations. Therefore the output is 8.2 rather than 8.20. Line 140 illustrates that VAL is necessary whenever a number assigned to a string variable is to be used with a numeric operator (+,−,*,/,^). Attempting to add a number to a string, such as C$ + 5, produces a ?TYPE MISMATCH ERROR.

The STR$ function is used to convert a number into a string. For example:

```
88 X$ =  STR$ (X)
```

If the numeric value of X is 27 then its corresponding string value is "27". This function can be used to determine the length of a number. The following program shows the correct coding of the STR$ function and its use in determining the length of a numeric entry:

```
10  INPUT "ENTER A NUMBER ";N
20  LET N$ =  STR$ (N)
30  LET L =  LEN (N$)
40  PRINT "THERE ARE ";L;" DIGITS IN ";N
```

Line 10 allows the user to enter a number. This number is converted to a string in Line 20, and the number of characters is then determined utilizing the LEN function in Line 30.

2.13 ON GOTO and ON GOSUB

ON GOTO is a multiple extension of the GOTO statement. The conditional statement is used to branch to one or more lines of a program:

```
10  INPUT "ENTER A NUMBER FROM 1 TO 3 ";N
20  ON N GOTO 400,500,600
30  PRINT "YOU DID NOT ENTER A 1, 2, OR 3": END
400  PRINT "ONE": END
500  PRINT "TWO": END
600  PRINT "THREE": END
```

In Line 10 the user is asked to enter a number from 1 to 3. The number entered is assigned to N. For example, if 1 is entered then N = 1. Line 20 sends program execution to Line 400, the first line number following the GOTO. Thus Line 20 will actually be executed as

```
20  ON 1 GOTO 400
```

However, if 2 is entered, N = 2 in Line 10 and program execution would branch to Line 500, because Line 20 would be equivalent to

```
20  ON 2 GOTO 500
```

If 3 is entered, N = 3 in Line 10 and program execution would transfer to Line 600. If an integer other than 1, 2, or 3 is entered then program execution drops to the following line (Line 30), and a message is printed indicating the failure to correctly follow the prompt in Line 10.

ON GOSUB is similar to ON GOTO, but it sends the program to a subroutine, the last line containing a RETURN statement:

```
10  INPUT "WHICH GREETING WOULD YOU LIKE (1 - 4)?";G
25  ON G GOSUB 1000,2000,3000,4000
35  END
1000  PRINT "HELLO": RETURN
2000  PRINT "GOOD MORNING": RETURN
3000  PRINT "HOW ARE YOU?": RETURN
4000  PRINT "HI THERE!": RETURN
```

Program control is dictated by the value of the input in Line 10. If G is 1, the program branches to the subroutine at Line 1000, the first line number following GOSUB. A 2 will cause a branch to Line 2000, the second line number after GOSUB, and so on. An integer less than 1 or greater than 4 will be disregarded by the ON GOSUB statement, and program control will drop to the following line (Line 35 END). The ON GOSUB statement differs from the ON GOTO statement in that program control returns to the statement following the ON GOSUB when a RETURN is encountered at the end of the subroutine.

PEEK and POKE 2.14

PEEK is a statement or command that displays the contents of a memory location. Apple's main memory consists of 65,536 bytes,

so there are 65,536 different memory locations. You can think of a memory location as a box or pigeonhole in which a value is stored. PEEKing a memory location returns the number stored at that address. Here is how you use PEEK to check the value at a memory location.

```
PRINT        PEEK (32)
0
```

The memory location 32 controls the left edge of the text window. A zero in this location is the default value, the value placed in location 32 when the computer is first turned on. The zero signifies that no left margin exists, and text will be printed in the first screen column.

POKE is a corresponding statement or command that actually changes the value of a specific memory location. The following command will create a left margin five columns wide on the text screen:

```
POKE      32,5
```

Any text now printed will be indented five spaces. To set the left margin back to its normal setting, simply put a zero back in memory location 32 as follows:

```
POKE      32,0
```

Since almost any memory location can be PEEKed or POKEd, and all information relative to the run of a program is stored in memory, be careful when using POKE statements. Inappropriate changing of values in specific memory locations can literally destroy your Applesoft program. So, be certain you have carefully checked the address of the intended memory location and the value you wish to place at this address before you use a POKE statement in a program. A list of useful PEEKs and POKEs may be found in a programming manual such as the *BASIC Programming Reference Manual* published by Apple Computer Inc.

ON LINE

1 Distinguish between numeric and string variables. List three examples of each.

2 How is LET used to assign a variable?

3 How are READ and DATA statements used to assign variables?

4 What are the two forms of the Applesoft INPUT statement?

5 How is a GET statement different from an INPUT statement?

6 Give one example of a GOTO statement and explain how it branches a program.

7 Explain how an IF-THEN statement can branch a program.

8 Why is a subroutine used in a program? Explain the use of GOSUB and RETURN in programming a subroutine.

9 How can a REMark be used in a multiple statement line?

10 Explain the use of a subscripted variable to assign the values in an array.

11 Write a routine that uses a FOR/NEXT loop to read data into a subscripted variable.

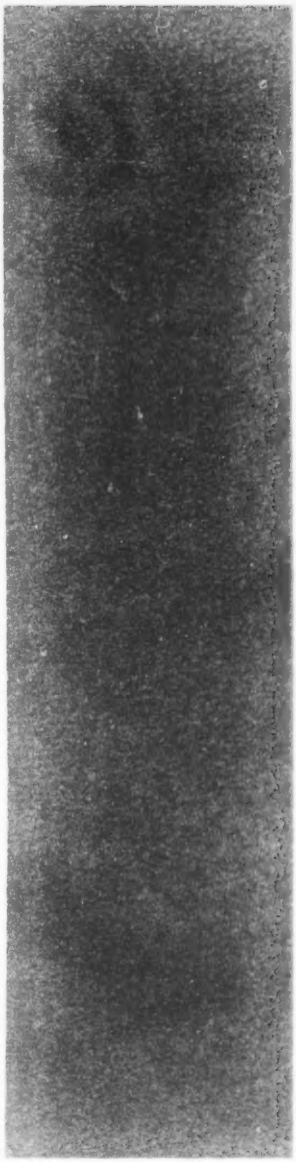
12 Explain how to use LEFT$, MID$, and RIGHT$ to print selected characters in the string JANUARY.

13 How can string functions be used to assign substrings to variables?

14 What is concatenation? Write a short program that concatenates two words assigned to two variables by INPUT statements.

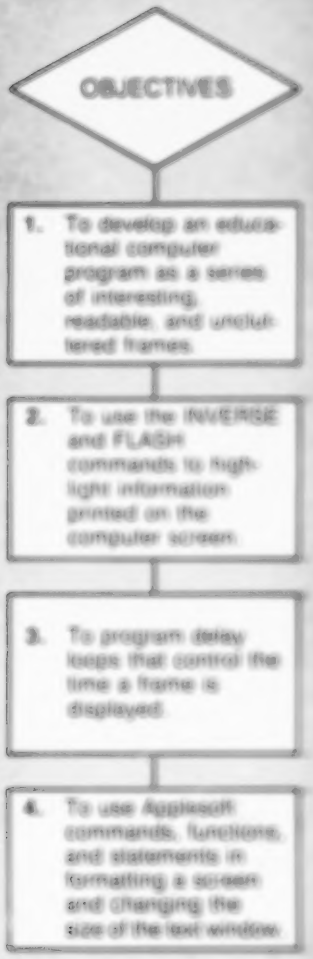
15 Give one example of the use of each of the following in Applesoft programming.

a. LEN b. CHR$ c. ASC d. VAL e. STR$
f. ON GOTO g. ON GOSUB



CHAPTER THREE

SCREEN FORMAT


OBJECTIVES
1. To develop an educational computer program as a series of interesting, readable, and uncluttered frames.
2. To use the INVERSE and FLASH commands to highlight information printed on the computer screen.
3. To program delay loops that control the time a frame is displayed.
4. To use Applesoft commands, functions, and statements in formatting a screen and changing the size of the text window.

This chapter discusses program development as a series of frames or screens displayed in sequence on the computer screen. Guidelines are given for developing a variety of screens that are interesting, ordered, easily readable, and uncluttered. Directions are given for using CALLs to format a screen by controlling movements of the cursor. The section on delay loops discusses their use as clocks that control the time a particular screen, such as a title screen, is displayed. Further, the programming of INVERSE and FLASH is explained in relation to the use of these Applesoft statements in enhancing the appearance of screen output by highlighting parts of text printed on the screen.

It is sometimes useful in writing an educational computer program to change the size of the text window; that is, the portion of the screen on which text is printed. The programming of this feature with POKEs is covered in Section 3.6. Other programming techniques covered in this chapter are creating screen borders and eliminating the input question mark when it detracts from the screen format.

3.1 Frames

An educational program should be designed as a series of frames. A frame in a computer program is the total amount of information presented on the microcomputer display screen at one time. Synonyms for frame are display, screen, and page. It may be helpful to think of a program as a filmstrip consisting of a consecutive series of pictures or frames. Each frame appears on the screen in succession, one at a time, as the filmstrip is drawn through a projector. Similarly, each frame of a computer program is displayed on the video screen as the program runs. The frames are separated by a HOME statement, which clears each frame from the screen before the next frame is printed. A computer program, however, departs from the filmstrip analogy in that the frames in the program need not be displayed in sequence. As the program runs, it may branch to frames at various points in the program in response to inputs by the user. For example, a program may give the user the option to display a menu at any time during the run of the program by pressing the M key. A menu frame lists the parts of a program and shows the keys the user must press to execute each part.

One hallmark of a good educational program is a variety of carefully programmed frames. The first frame in the program should be the title frame, which prints the name of the program and perhaps

other pertinent information, such as the name of the programmer. The last frame should contain an ending statement. It could very well be a character graphic design of the phrase "The End," or it could be a more imaginative closing in which a curtain is drawn across the screen followed by a simple printing of "Finis." (See Section 6.7 for a discussion of character graphics in programming.) If the program is to be menu driven, use a frame to display the menu and write the program so that the student can return to the menu frame at will. If your program contains questions or exercises, you can achieve clarity in the presentation by using one frame for each question or exercise.

Figure 3.1
This frame in the tutorial program, End Punctuation, prints information pertaining to punctuating the end of a sentence.

```
800  PRINT "THERE ARE THREE PUNCTUATION MARKS THAT "
810  PRINT "CAN BE USED AT THE END OF A SENTENCE."
900  VTAB 12
910  PRINT  TAB( 5);"PERIOD"; TAB( 25);"."
920  PRINT
930  PRINT  TAB( 5);"QUESTION MARK"; TAB( 25);"?"
940  PRINT
950  PRINT  TAB( 5);"EXCLAMATION MARK"; TAB( 25);"!"
```

Avoid screen clutter. Generally, do not print more than six lines on the screen at one time and use PRINT statements to leave adequate spacing between the lines. Moreover, there should be no upward scrolling of text as it is displayed on the screen.

What length of time should a frame be displayed on the video screen? You can control the time with a delay loop (Section 3.4) that creates a pause before the HOME statement is executed and the screen is cleared. An example of a delay loop is given in Line 100 of the following listing. The line contains a FOR/NEXT loop in two statements separated by a colon. P is incremented by 1 each time through the loop. After 5000 iterations in which there are no calculations and no output to the screen, the program drops out of the loop and goes to Line 110, at which point the screen is cleared by the HOME statement.

```
100  FOR P = 1 TO 5000: NEXT P
110  HOME
```

Usually it is best to have the frame displayed as long as the student wants it on the screen. You can program this feature in two ways: (1) requiring an input from the student before the program moves to the next frame; (2) printing PRESS SPACE BAR TO CONTINUE at the bottom of the frame and using a GET statement to accept the keystroke. In both cases the frame is on the screen until the student types a response on the keyboard that, in effect, gives the student control of the program run. Specific directions for incorporating user control in your program are given in Section 4.2.

An example of a frame that presents a learning objective for junior high school science is given in Figure 3.2. The program lines that produce the frame are listed below it. Note the use of the HOME statements in Lines 10 and 50 to clear the screen and the use of a delay loop in Line 40 to control the time the frame is displayed.

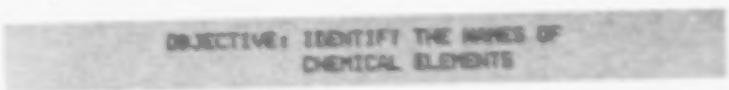


Figure 3.2
Example of a frame displaying an objective.

```
10  HOME
20  VTAB 10
30  PRINT "OBJECTIVE:IDENTIFY THE NAMES OF"
32  PRINT
34  PRINT  TAB( 12)"CHEMICAL ELEMENTS"
40  FOR P = 1 TO 3000: NEXT P
50  HOME
```

Figure 3.3 shows a frame that lists a table of information for use in running an educational computer program pertaining to animal heartbeats. As indicated in the listing that follows (Lines 1390–1480), the names of the animals are printed through use of string variables that have previously been assigned the names of the respective animals. A student using the program may study the data as long as desired before pressing the Space Bar to continue. Printing PRESS SPACE BAR TO CONTINUE and advancing the program to the next frame are handled by a separate subroutine as described in Section 4.2.

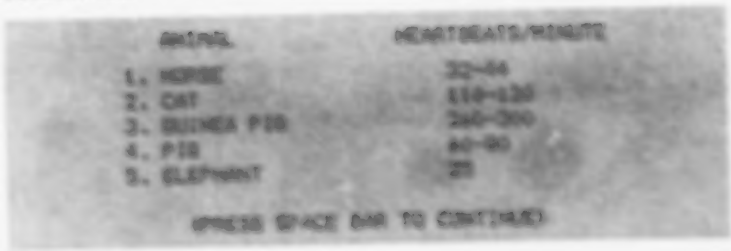


Figure 3.3
Frame showing a table of data.

```
1390  REM  SUBROUTINE TO PRINT TABLE OF ANIMAL HEARTBEAT RATES
1400  HOME : VTAB 9
1410  PRINT "ANIMAL              HEARTBEATS/MINUTE"
1420  PRINT "------              ----------------"
1430  PRINT "1. ";H$; TAB( 25)"32-44
1440  PRINT "2. ";C$; TAB( 25)"110-130"
1450  PRINT "3. ";G$; TAB( 25)"260-300"
1460  PRINT "4. ";P$; TAB( 25)"60-80"
1470  PRINT "5. ";E$; TAB( 25)"25"
1480  RETURN
```

Figure 3.4 shows a frame in which a multiple choice question is printed. Note the question mark in the bottom center of the frame that signals the need for an input by the student. Since the input is required before the program moves to the next frame, the student has control in determining how long the question is displayed on the screen before the HOME statement in Line 580 of the subsequent listing is executed. After an input is made by the student, the HOME statement clears the screen for printing of the next frame.

Figure 3.4
This frame displays a multiple choice question.

```
500  HOME
510  VTAB 7
520  PRINT "5. WHICH BLOOD VESSELS ARE THE SMALLEST?"
530  PRINT : PRINT
540  PRINT  TAB( 3);"A. ARTERIES  B. VEINS  C. CAPILLARIES"
550  PRINT : PRINT
560  HTAB 19
570  INPUT ANS
580  HOME
```

In the interest of clarity, an input check has not been included in this program. However, as explained in Section 4.5, it would be important to include an input check that limits the entry to three possible choices: A, B, C. In other words, the acceptable input should be limited to one of these three letters and nothing else. Moreover, as explained in Section 4.6, a GET statement may be used to accept the entry (A, B, or C) and assign the value to ANS.

Formatting a frame so it is pleasing to the viewer's eye will give your program a professional look and make it easier to read. Techniques for increasing the visual interest of a frame, such as using INVERSE and FLASH statements, are discussed in subsequent sections. Moreover, in printing a frame you may wish to include a border around the edge of the screen and change the size of the window in which text is printed. Directions for changing the size of the text window are given in Section 3.6.

1 Explain why the format of a program is an important factor in its development?

2 Define the term "frame" and explain how HOME is used to create frames in a BASIC program.

3 What is a menu frame?

4 How is an IF-THEN statement used to branch to a frame? Give one example.

5 What is a delay loop, and how is it used to display a frame on the computer screen?

6 What is meant by student control of the run of a program?

3.2 Screen Displays

Successful presentation of your educational program is highly dependent on its visual appearance. In a well-structured presentation, screen displays will go unnoticed, allowing total concentration and attention on the substance of the program. Screen displays must be orderly and easily readable; cluttered, nonsymmetric presentations divert the user's attention from content to format. Although no firm rules exist for organizing printed information on the screen, some recognized guidelines, dealing with the amount and location of material on the monitor at one time, are generally followed.

When information is presented, it is best displayed with a maximum of six printed lines. Starting a few lines from the top of the

screen and double spacing will produce an easily readable professional look. Contrary to textbook formats, sentences need not originate at the left margin; in fact, center justification is perfectly acceptable and, in many instances, recommended in computer presentations. Lines should end cleanly, words never wrapping around to a second line. Specific phrases such as square root, United States, proper noun, and names like Bruce Springsteen should also be placed on a single line.

To attain this desired result, programmers use PRINT statements and TAB functions. PRINT is used to leave a space between printed lines. Both TAB(X) and HTAB X will indent a phrase the number of places specified by X. VTAB X is used to establish the vertical placement of the line to be printed. Run the following program to get a better idea of just how PRINT statements and TAB functions operate.

```
5  HOME
6  PRINT : PRINT
8  PRINT  TAB( 9);"--INFORMATION PLEASE--"
10  PRINT : PRINT : PRINT  TAB( 5);"NAME:"
12  PRINT : PRINT  TAB( 5);"AGE:"
14  PRINT : PRINT  TAB( 5);"PHONE #:"
16  PRINT : PRINT  TAB( 5);"FAVORITE SUBJECT:"
20  HTAB 11: VTAB 6: INPUT N$
22  HTAB 10: VTAB 8: INPUT A$
24  HTAB 14: VTAB 10: INPUT P$
26  HTAB 23: VTAB 12: INPUT FS$
28  HTAB 15: VTAB 15: PRINT "THANK YOU"
30  END
```

An explanation of the various program lines should provide the fundamentals necessary to create your own screen displays.

5 Clears the screen and sets the cursor to the top of the screen.

6 Moves the cursor down two lines.

8 The TAB positions the cursor so that the output will begin in the ninth column, centering the phrase.

10-16 All similar program lines. The extra PRINT statements provide double spacing, and the TAB(5) sets a left margin five columns from the edge of the screen.

20-26 The HTABs set the INPUT prompt to the appropriate location following the corresponding printed phrase, and the VTABs place the prompt on the proper line.

The INPUT statements allow the user to enter the requested information and assign this information to the various string variables.

28 HTAB 15 centers the phrase to be printed, THANK YOU, and VTAB 15 enables the message to be printed on the fifteenth screen line.

TAB functions can insure both proper line spacing and line justification. Although more programming time and computer memory are required to format a screen display in this manner, the results produce a polished display that does not detract from the educational intent of the program.

Another formatting consideration is the standardized location of certain information that repeatedly appears on the screen. If, for instance, your program includes quiz questions, they should be presented in the same screen area. The scenario generally originates with a question being asked and a response given. After the screen is cleared, the second question should appear at the same screen location with the response prompt likewise being located in a standard position. VTABs and HTABs, along with subroutines, can guarantee this customized and consistent format. Although no brief routines demonstrating these techniques will be listed at this time, many of our examples will follow the aforementioned procedures. Take note of the ease with which the output of these routines are read and make every effort to incorporate such techniques in your educational programs. The success of your creation may very well hinge upon the way in which it is displayed!

```
200  REM  PRINT TITLE
210  HOME : VTAB (9)
220  PRINT  TAB( 9)"**********************"
230  PRINT  TAB( 9)"*                    *"
240  PRINT  TAB( 9)"*                    *"
250  PRINT  TAB( 9)"*                    *"
260  PRINT  TAB( 9)"**********************"
270  SPEED= 25
280  VTAB 11: HTAB 11
290  PRINT "ANIMAL HEARTBEATS"
300  SPEED= 255
310  FOR P = 1 TO 2000: NEXT P
```

Figure 3.5
Formatting of the screen display of the title for Animal Heartbeats. Line 280 tabs the printing of the title within the border formed by the asterisks.

ON LINE 3.2

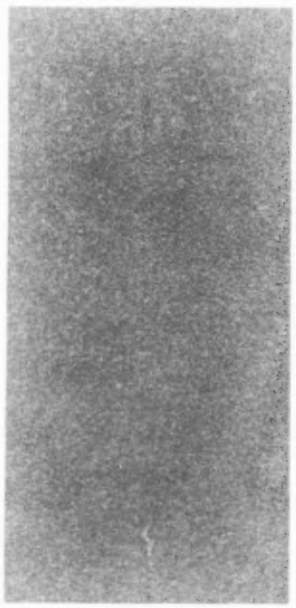
1 Use HOME, VTAB, and HTAB to center the phrase HAPPY NEW YEAR on the screen.

2 Write a program that prints the six New England states (Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island) on the screen. Experiment with various spacings and margins, concluding with what you would consider an appropriate screen display of this information.

3 Write a program that prints your name, room number, and school in the center of the computer screen.

4 Write a program that prints the names of five students and their grades.

5 Display the Preamble to the Constitution. End all lines cleanly and keep appropriate phrases on the same line. "We, the people of the United States, in order to form a more perfect Union, establish justice, insure domestic tranquility, provide for the common defence, promote the general welfare, and secure the blessings of liberty to ourselves and our posterity, do ordain and establish this Constitution for the United States of America."



Screen Formatting CALLs 3.3

The Apple computer contains a series of built-in routines that can be accessed by referring to the appropriate memory locations. The instruction that executes these machine language subroutines is a CALL. Like most program statements, a CALL may be placed anywhere in a program and will be executed when read. As with the GOSUB statement, the program returns to the next statement in the program upon completion of the CALL routine. An example of a CALL statement is

```
10  CALL  - 936
```

This statement is identical to the HOME statement that clears the screen and returns the cursor to the upper left corner. Try inserting this CALL where a HOME statement exists in a program and note the effect when the program is run.

If all CALLs duplicated existing BASIC commands and statements there would certainly be no need for their mention here. There are, however, certain programming situations in which some CALLs can

facilitate specific cursor positioning and screen changes with greater programming ease than other BASIC statements can. The following is a list of some useful screen formatting CALLs along with a description of the corresponding routine.

- CALL −868 Clears the screen from the cursor to the end of the line
- CALL −912 Scrolls the text up one line feed
- CALL −922 Moves the cursor down one row
- CALL −958 Clears the screen from the cursor to the bottom row
- CALL −998 Moves the cursor up one row
- CALL −1008 Moves the cursor to the left one column
- CALL −1036 Moves the cursor to the right one column

Before exploring the benefits of using such CALLs in educational programs, you should enter and run the following brief routines on your computer. The intent of these routines is to help you visualize the results of these CALLs when inserted into programs.

```
10  FOR I = 1 TO 350
20  PRINT I;
30  NEXT I
40  VTAB 10: HTAB 5
50  CALL  - 958
60  END
```

This program fills the screen with the numbers from 1 to 350 and then positions the cursor at the tenth row and fifth column. The CALL −958 in Line 50 clears all text beyond the cursor to the bottom of the screen. Changing Line 50 to CALL −868 will result in the clearing of all characters on Row 10 to the right of the cursor. Try it!

The second exploratory routine allows you to examine the resulting cursor position determined by various CALLs.

```
10  HOME
20  VTAB 12: HTAB 20
30  GET K$
40  PRINT K$;
50  CALL  - 922
60  GOTO 30
```

Run this program and press any key several times. After noting the placement of the output on the screen, press Control-Reset to halt the run of the program. This routine can be used to study the effect

of many of the screen formatting CALLs. Simply change Line 50 to one or more of the CALLs listed at the beginning of this section. Some of the more interesting results are generated by the following:

```
50  CALL  - 912
50  CALL  - 1036
50  CALL  - 1008: CALL  - 1008
50  CALL  - 998
50  CALL  - 998: CALL  - 1008
50  CALL  - 958: CALL  - 998
```

Once you have familiarized yourself with the cursor routines activated by these various CALLs, you can insert one or more of them in the appropriate place in your educational program.

The following routine offers the programmer a unique way to have the user's input displayed on the screen. The student is asked to input his or her name, and as this is done it is displayed vertically! Enter the following routine, which employs a CALL −1008 and a CALL −922, and note the difference from the standard horizontal output used so frequently in computer programs.

```
10  PRINT "ENTER YOUR NAME"
15  PRINT "PRESS RETURN WHEN DONE"
20  GET K$
30 N$ = N$ + K$
40  PRINT K$;
50  CALL  - 922
60  CALL  - 1008
70  IF K$ <  >  CHR$ (13) THEN 20
80  PRINT : PRINT "HELLO, ";N$
90  END
```

Lines 10 and 15 ask the user to enter his or her name and press Return when done. The GET statement in Line 20 takes the first keyboard stroke and assigns it to K$. Line 30 concatenates it with any previous entry. Line 40 prints the character accepted by Line 20. Line 50 moves the cursor down one row, and Line 60 moves the cursor back one column, creating the vertical input routine. Line 70 checks to determine if the Return key was pressed. If it has, Line 80 prints the word HELLO followed by the name entered. If Return is not pressed, control goes to Line 20, where another keyboard entry is received.

Another example using CALL statements follows. It uses three cursor positioning CALLs to produce a computer routine that can be used to improve a student's keyboarding skills. Enter the following

program into your computer and run it. Note how long it takes you to complete the keyboarding exercise generated by this program.

```
10 A$ = "FWUNXLPGTABYVOSZIDH"
20  FOR I = 1 TO  LEN (A$)
30  PRINT  MID$ (A$,I,1);
40  CALL  - 1008: CALL  - 922
50  GET K$
60  IF K$ <  >  MID$ (A$,I,1) THEN 50
70  PRINT K$;
80  CALL  - 998
90  NEXT I
95  END
```

Line 10 assigns a string of letters to A$. Line 20 starts the FOR/NEXT loop that increments the position of the letter in A$ that will be displayed on the screen by Line 30. The output generated by Line 30 is a single letter selected from A$. Line 40 positions the cursor directly beneath the character generated by Line 30. The GET statement in Line 50 assigns the character entered via the keyboard to K$. If this character is not identical to that printed by Line 30, then the student must enter another character. After the matching character is input, the cursor is now moved up one screen line where the next letter selected from A$ is printed.

The preceding program illustrates the independence of the cursor positioning CALLs from any particular row and column coordinates set with VTAB and HTAB statements. This program can be run from any screen location, and the length of A$ will not affect the running of the program. Try "homing" the screen before the program is run. Then list the program and run it. Creation of a similar program using HTABs and VTABs probably would not be as independent as this version.

Experiment with the length of A$. Assign a string of more than 40 characters to A$ and note the cursor positioning when the forty-first character wraps around to the next screen line. What would occur if VTABs and HTABs were used in the program instead of the cursor positioning CALLs?

Playing no less an important role in this routine than the CALLs, however, are the semicolons in Lines 30 and 70. Cursor positioning CALLs lose their effectiveness when the semicolons are omitted. A semicolon following a PRINT statement positions the cursor one column to the right of the output, whereby appropriate CALLs can manipulate that cursor position to attain whatever location is desired.

The final CALL routine illustrates the use of the text clearing CALLs, CALL −868 and CALL −958. Enter the following program

into your computer and run it several times. Try a variety of numerical inputs, including some that do not satisfy the stated conditions.

```
10  HOME
20  VTAB 6: PRINT "ENTER YOUR VERBAL S.A.T. SCORE ";
30  CALL  - 868
40  INPUT V
50  IF V < 200 OR V > 800 THEN  GOSUB 500: GOTO 20
60  CALL  - 958
70  VTAB 12: PRINT "ENTER YOUR MATH S.A.T. SCORE ";
80  CALL  - 868
90  INPUT M
100  IF M < 200 OR M > 800 THEN  GOSUB 500: GOTO 70
110  CALL  - 958
120  END
500  VTAB 23: PRINT "THAT IS NOT A VALID SCORE"
510  RETURN
```

This routine asks the user to enter two S.A.T. scores. If the scores do not fall within the accepted range, then the message THAT IS NOT A VALID SCORE is printed at the bottom of the screen and control returns to the line containing the question. The CALL −868 in Line 30 erases any previous response to the question in Line 20, allowing the user to reenter a score without seemingly writing over his or her previous unacceptable response. This routine appears again in Lines 70 and 80.

The CALL −958 in Line 60, and again in Line 110, erases the error message at the bottom of the screen. If an invalid number is entered in Line 40, the message from Line 500 would appear. However, when the user reenters a score, this time valid, the error message is erased by the CALL −958. These CALLs provide an efficient method of accepting input and displaying the corresponding output.

Though CALLs are used less frequently than most other BASIC statements, they can play a significant role in creating accurate, useful, and diversified programming routines. The examples presented in this section illustrate but a few of the many applications of screen formatting CALLs. In addition to the examples presented here, the routines in Section 7.4 also employ many of the aforementioned CALLs to format numerical output. As you develop and expand your educational programs, you will find these CALLs to be the perfect prescription for many of your screen formatting needs.

ON LINE 3.3

1 What is a CALL statement?

2 How can CALL statements be used in formatting a program?

3 Write a program that prints a question on the screen and then moves the cursor one column to the right for printing an input question mark.

4 Enter the program on page 82 into your computer. Try changing Line 50 to each of five different CALLs. What is the effect of each CALL on the run of the program?

5 Write a program that prints a multiple choice question and uses CALL statements in formatting the screen.

3.4 Delay Loops

Some frames in your educational computer program may present more detail than others. Consequently you may wish to provide the student more time to read the text displayed on the screen. You can provide this time by using a FOR/NEXT delay loop. To create pauses between screen displays, simply insert, at the appropriate locations in the program, a FOR/NEXT loop that contains no statements between the FOR and NEXT statements. We call this a "do nothing" loop. Although it seemingly does nothing, time is still required for the loop to go round and round in the computer looking for something to do.

Delay loops can serve as clocks or timers in your programs: the greater the number of iterations performed by the loop, the greater the time delay. Obviously some computers will execute these loops faster than others. Our experiments show that the Apple II+ performs approximately 740 iterations or loops in 1 second. That is, the Apple requires about 1 second to execute the following command: FOR P = 1 to 740:NEXT P.

Now we have a benchmark that we can use to create pauses of different durations. To provide a 2 second delay, multiply 740 by 2, which is 1480, and include this number in the FOR/NEXT loop. For a 3 second delay multiply 740 by 3, and so on.

```
10  FOR P = 1 TO 3700: NEXT P
20  PRINT "DONE"
```

After entering the program, type RUN and use the second hand of your watch or a stopwatch to time the execution. Start timing the instant you push the Return key. The word DONE should be printed about 5 seconds after the program is started because 3700 is 5 times 740.

Although the most apparent, and perhaps appropriate, use of the delay loop is to provide necessary pauses between screen delays in your educational programs, the delay loop can also be used to add some interesting effects to the program. Examine the following lines that introduce a program on the use of nouns in English.

```
10 X = 0
20  PRINT "N";
30  GOSUB 1000
40  PRINT "O";
50  GOSUB 1000
60  PRINT "U";
70  GOSUB 1000
80  PRINT "N";
90  GOSUB 1000
100  PRINT "S"
105  END
1000  FOR P = 1 TO 3700 - X: NEXT P
1020 X = X + 740
1030  RETURN
```

When the program is executed, the letter N is printed, followed in 5 seconds by the letter O, followed in 4 seconds by the letter U, followed in 3 seconds by the letter N, followed in 2 seconds by the letter S. The result is the word NOUNS printed letter by letter at an increasingly rapid rate, thus adding a nice touch to the program.

The crux of the timing delay in this program is in the subroutine beginning at Line 1000. Note the use of a variable X in the FOR/NEXT loop. As X increases, the time delay decreases. Also note that in Line 1020, X is increased by 740, the amount required in a FOR/NEXT loop to provide a delay of 1 second. Consequently, each successive call to the subroutine results in a time delay 1 second less than the delay created by the previous call.

To use time delays in your educational program, include a FOR/NEXT loop at the appropriate location and use a number in the loop that is large enough to provide the time required by students

to read the text being presented. However, don't overuse the time delay feature. Including this feature too often will create a boring program.

---

1 What is a delay loop, and how is it used to create a pause in the run of a program?

2 Why is a delay loop referred to as a "do nothing" loop?

3 Where would you use a delay loop in an educational computer program?

4 a. Write a delay loop having 3000 iterations.
b. Change the loop so that the time delay will be reduced by one third.
c. Change the loop so that the time delay is doubled.

5 Write a short program with a delay loop that prints the title to a program. Enter the program into your computer. Run the program and adjust the loop so that the title is displayed on the screen for 3 seconds.

6 Describe two important uses of delay loops in educational computer programs.

7 Write an educational computer program that uses the delay feature in a way not discussed in this chapter.

8 Write a delay loop that provides approximately a 10 second delay in a program run.

9 Write an educational computer program that presents a spelling word on the screen for a brief time followed by the user entering the word spelled correctly. Use a delay loop to display the word on the screen for a progressively longer interval each time it is misspelled by the user. The user should be given five chances to spell the word correctly.

Text on the Screen 3.5

Once you have decided on the amount of information to display on the screen at one time and the location of this information, you then have the option of putting some or all of it in normal, inverse, or flashing modes. Depending on the computer, you may also have the choice of using uppercase or lowercase letters.

To activate the inverse mode, insert the statement INVERSE in your program. From then on, all printed material appearing on the video screen will be displayed in inverse fashion—that is, black letters on a white background. To return to the standard display mode, insert a NORMAL statement in your program.

Enter this program and run it to see how the INVERSE and NORMAL statements can highlight key points in a sentence.

```
1  HOME
5  VTAB 8
10  PRINT "USE WORDS TO:"
15  VTAB 10
20  PRINT "EXPRESS THE NUMBERS ";
25  INVERSE
30  PRINT "ZERO";
35  NORMAL
40  PRINT " AND ";
45  INVERSE
50  PRINT "ONE";
55  NORMAL
60  PRINT " WHEN..."
```

In most instances, the phrases and sentences included in educational computer programs are definitions, directions, or brief explanations of concepts. They tend to be precise and to the point. As illustrated, the use of INVERSE can highlight and thereby emphasize the principal word or words in the sentence. Notice how the key words ZERO and ONE are printed when Lines 30 and 50 are executed. In both cases the INVERSE statement precedes the key word, and the NORMAL statement follows. Other observations in this program include the semicolons after any printed phrase and blanks preceding and following certain words. Semicolons cause the material to be printed as a sentence; the spaces within the strings create spacing between words in the sentence.

Though multiple statement lines are generally not recommended, the preceding program may be rewritten as follows:

```
1 HOME
5 VTAB 8: PRINT "USE WORDS TO:"
10 VTAB 10: PRINT "EXPRESS THE NUMBERS ";: INVERSE :PRINT "ZERO";: NORMAL
   : PRINT " AND ";: INVERSE : PRINT "ONE";:NORMAL :PRINT " WHEN..."
```

Many programmers find it convenient to use one program line for each line that is to be viewed on the screen. This practice allows for a one-to-one program line–screen line correspondence that facilitates the location of such outputted lines in the program. Though more difficult to edit when adjustments are necessary, multiple statement program lines in instances such as this prevent a needless rambling within the program and establish a program line–screen line identification system that can prove most convenient when debugging is necessary. To create a multiple statement program line, type a colon between each statement on the same line.

The INVERSE statement can definitely enhance the appearance of a frame and highlight sections of it. If used for stressing important facts, its inclusion should be limited. One or two items per frame is adequate and the word or phrase should warrant highlighting; overusing highlighting can dilute its effectiveness. Every frame does not need an INVERSE statement.

INVERSE may also be used to underscore a key word or phrase. Run the following program lines. The ^ ^ ^ ^ underscores ZERO and ONE.

```
62  HOME
65  VTAB 14: PRINT "EXPRESS THE NUMBERS ZERO AND ONE WHEN.."
70  INVERSE
75  HTAB 21: PRINT "^^^^";: HTAB 30: PRINT "^^^^": NORMAL
```

Combining these program lines with those previously listed will allow you to compare both displays. The caret (^) is used in the INVERSE underscoring for variety only; they are not mandatory.

A second output mode can be created by the FLASH statement. An alternating of the INVERSE and NORMAL modes, FLASH can be used to catch the immediate attention of the user. It is especially useful when specific input is required. For a visual demonstration of how FLASH can be incorporated into an educational program, clear the screen, enter, and run the following program lines.

```
10  PRINT "ENTER A NUMBER GREATER THAN 10"
15  INPUT N
20  IF N > 10 THEN  END
25  PRINT : PRINT : PRINT
30  FLASH
```

```
35  HTAB 6: PRINT "INVALID ENTRY-PLEASE REENTER"
40  NORMAL
45  PRINT : GOTO 15
```

Obviously, one must enter a number less than or equal to 10 for this example to be illustrative. When Line 30 is executed, all further output will be displayed in flashing mode; thus the entire phrase in Line 35 flashes on the screen. The NORMAL statement in Line 40 terminates the flashing mode and resets the output to the standard format. You may want to delete Line 40 and run the program a few times, continually entering invalid numbers.

Figure 3.6
Line 1000 in The Number Machine uses FLASH in the printing of THINKING at the eighth row, Column 11.

The FLASH and INVERSE statements may be used interchangeably. However, INVERSE is best used to highlight a specific word or phrase, whereas FLASH is at its best when a warning or error message must be displayed. Highlighting with the FLASH statement can be quite annoying to the user and is not recommended for that purpose. No more than one flashing statement need be on the screen at one time.

Don't overuse the INVERSE and FLASH alternatives. The standard text display is more than adequate in most frames; INVERSE and FLASH should be limited to special situations. Any text forms that detract from the content should be avoided. Programming experience will help you decide how often to use INVERSE and FLASH.

If your computer can print lowercase letters, you may decide to incorporate this feature when creating your program. But consider: (1) All programming commands may have to be entered in capital letters. Although the Apple IIc and newer Apple IIe (65SC02 microprocessor) accept commands and variables typed in lowercase letters, the Apple II+ and older Apple IIe reject this format. This means INVERSE and FLASH must be capitalized in the line statements. (2) The computer on which your program is run must be able to produce lowercase letters. If you decide to use both uppercase and lowercase letters, then do so consistently. Standard rules of capitalization apply, and any deviation should be avoided.

In summarizing, it is wise to always keep in mind that content has priority. Any distraction must be warranted and should only be used to convey specific meaning in a unique but informative style. Although moderation in the incorporation of these alternative methods of displaying text on the screen does not insure proper use, moderation will increase the likelihood that your program will be enhanced by these different styles.

1 Compare the appearance of text printed in NORMAL, INVERSE, and FLASH modes.

2 What kinds of information should be highlighted on the computer screen? Give three examples.

3 Display the following sentence on the computer screen, using INVERSE to highlight the word FRIDAY.

THE ASSIGNMENT IS DUE ON FRIDAY

4 Use INVERSE mode to underscore the phrase WHITE BIRCH in the following sentence: "Algonquins used the bark of the white birch to build canoes."

5 Print the word COMPUTER on the computer screen, flashing the vowels and keeping the consonants in NORMAL mode.

3.6 Text Window

All 24 lines and 40 columns of the Apple II display screen are normally used when in the text mode. However, you can change the size and position of the text window, that portion of the screen on which text is displayed. While a program is running, the size of the window may be changed with POKE statements. For example, suppose you want to display a heading at the top of the screen and to print several sequential screens of text underneath, clearing the screen, but not the heading, between each printing. You can program this format so that the heading will be retained each time the screen is cleared and new information is presented. POKE 34,x is used to shrink the text window down by lowering the top margin of the screen. The x after the comma specifies the number of rows

down at which the margin is set. For example, POKE 34,0 sets the margin at the top of the screen. POKE 34,10 sets the upper margin of the text window 10 rows down the screen.

The following program shows the use of POKE 34,3 to shrink down the text window so that a heading is maintained at the top of the screen. To see how this is done, enter the program into your computer and run it.

```
100  HOME
105  PRINT : PRINT
110  PRINT "BASEBALL STATISTICS"
120  POKE 34,3
130  GOSUB 200
140  PRINT "BATTING AVERAGE = .302"
150  GOSUB 200
160  PRINT "RUNS BATTED IN = 40"
170  GOSUB 200
180  PRINT "HOME RUNS = 5"
185  TEXT : END
200  FOR P = 1 TO 2000: NEXT P
220  HOME
225  VTAB 12
230  RETURN
```

Line 100 clears the screen. The PRINT statements in Line 105 leave two blank lines above the heading, BASEBALL STATISTICS, printed at Row 3 by Line 110. POKE 34,3 in Line 120 protects the heading from being cleared by a subsequent HOME statement. When the HOME statement in the subroutine (Lines 200–230) is executed, BASEBALL STATISTICS remains on the screen and the text printed below, such as BATTING AVERAGE = .302, is cleared from the screen. A different baseball statistic is printed (Lines 140, 160, 180) each time the screen is cleared. POKE 34,3 lowers the text window three rows. How can you get back to a full text window after this statement has been executed? Add a line with a TEXT statement to your program where you want to return to the full text window. Line 185 resets the text window before the program ends.

In similar fashion you can use POKEs to change the left margin of the text window (POKE 32,x), width of the window (POKE 33,x), and bottom margin (POKE 35,x). Using these POKEs, you can change the size of the text window for special printing effects. The values specified by x are given in Table 3.1.

Table 3.1
POKEs used to change the size of the text window

POKE	*Rows or Columns*
32 Left margin	0–39
33 Width	1–40
34 Top margin	0–23
35 Bottom margin	0–23

Here is a program that reduces the size of the text window all around and centers it on the screen.

```
90  HOME
100  POKE 32,9
110  POKE 33,22
120  POKE 34,6
130  POKE 35,14
135  HOME
140  FOR I = 1 TO 30
150  PRINT "REDUCED SCREEN SIZE ";
160  NEXT I
170  TEXT
```

Type the program and run it to observe the visual effect. You will see that the output, REDUCED SCREEN SIZE, is displayed only within the reduced text window. You can follow the run of the program through the line by line explanation that follows.

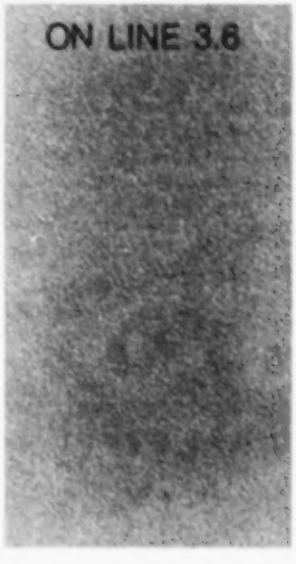
100 Sets the left margin of the text window at Column 9.
110 Sets the width of the window to 22 columns.
120 Sets the top of the window at Row 6.
130 Sets the bottom of the window at Row 14.
135 Clears the screen within the text window.
140 Programs the counter in the FOR/NEXT loop for 30 iterations.
150 Prints the message on the screen.
160 Sends the program back to Line 140 for another run through the loop and repeated printing of the message by Line 150. The looping is completed when I = 30.
170 Resets the text window.

After you have run this program go ahead and experiment. Change the values in the POKEs that specify the rows and columns for the text window. Run the program and observe the change in

size and location of the text window in which you see the video display. Caution: Do not use values other than those given in Table 3.1.

```
1570  REM  SUBROUTINE TO BEEP HEARTBEATS
1580  HOME : VTAB 9
1590  PRINT  TAB( 12)"????????????????"
1600  PRINT  TAB( 12)"?? HEARTBEAT ??"
1610  PRINT  TAB( 12)"????????????????"
1620  POKE 34,14: REM  SET TOP MARGIN OF TEXT WINDOW TO PREVENT SCROLLIN
      G OF LINES 1590-1610
1630  REM  HEARTBEAT RATE COUNTING AND TIMING LOOPS
1640  FOR T = 1 TO R
1650  FOR P = 1 TO X: PRINT "": NEXT P
1660  PRINT 86
1670  NEXT T
1680  TEXT : REM  RESET TEXT WINDOW
1690  HOME
1700  RETURN
```

Figure 3.7
Line 1620 uses a POKE to prevent scrolling of Lines 1590–1610. TEXT in Line 1680 resets the text window.



1 What four POKEs control the size of the text window? How does each POKE change the window?

2 How does a TEXT statement affect the size of the text window?

3 Write a program that prints a title at the top of the computer screen and a question in the center of the screen. Add a POKE statement that sets the top margin of the screen so that only the title remains when a HOME statement is executed.

4 Revise your program so the title is printed at the bottom of the screen. Change the POKE statement so that the title remains on the screen when the HOME statement is executed.

5 Write a program that reduces the text window to a size 10 columns wide and 10 rows high in the center of the

screen. Add lines to the program that will fill the text window with repeated printing of EDUCATIONAL PROGRAM.

3.7 Screen Borders

An aesthetic feature you can add to the display screen is a border that encloses and reduces the text window. You may want to place a border on the screen when displaying a title, asking a question, printing the results of a quiz, or framing some other kind of output for which a border will enhance the visual effect. The following program uses the letter X to create a screen border. Type the listing on your computer and run it. You will see a border around the screen five X's wide on the side and four X's high at the top and bottom.

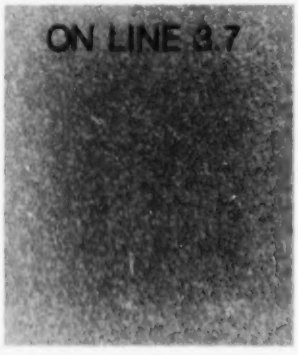
```
12  HOME
15  GOSUB 300
62  VTAB 5
65  FOR I = 1 TO 12
70  PRINT "XXXXX"; TAB( 36);"XXXXX";
80  NEXT I
90  GOSUB 300
150  VTAB 11: HTAB 11
160  PRINT "EDUCATIONAL PROGRAM"
170  VTAB 20
200  TEXT : END
300  FOR I = 1 TO 4
310  FOR J = 1 TO 40
320  PRINT "X";
330  NEXT J
340  NEXT I
350  RETURN
```

When the screen border program runs, the screen is cleared and Line 15 calls the subroutine beginning at Line 300. The subroutine (Lines 300-350) prints a border of four rows of X's across the top of the screen, 40 columns wide. Line 62 moves the cursor to Row 5 just below the top border. The FOR/NEXT loop in Lines 65–80 prints a border of five X's at the left of the screen and a border of five X's at the right. Line 90 calls the subroutine at Line 300 again to print a border of four lines of X's across the bottom of the screen. Lines 150 and 160 cause the printing of the title, EDUCATIONAL PROGRAM, in the center of the screen, and Line 170 moves the cursor to the bottom of the screen, outside the border.

After the border has been created, it may be kept on the screen during the subsequent printing of text by reducing the text screen by using POKEs. Add the following lines to the program and run it again.

```
171  POKE 32,9
172  POKE 33,22
174  POKE 34,6
176  POKE 35,14
177  FOR P = 1 TO 2000: NEXT P
180  HOME
185  FOR I = 1 TO 20
190  PRINT "REDUCED SCREEN";
195  NEXT I
```

The POKEs (Lines 171–176) reduce the screen dimensions within the printed border. Line 177 is a delay loop. When the HOME statement in Line 180 is executed, the smaller-sized screen inside the border is cleared, and the border remains on the screen. The REDUCED SCREEN message is printed repeatedly inside the border by the FOR/NEXT loop that follows (Lines 185–195). Note that the output from the program is restricted to the smaller window. Experiment! Change the design of the border by printing another character instead of X. For instance, replace each X in Lines 70 and 320 with an *. You might want to make a border of $ if the program pertains to money. Or use an ! for the border if the output is to be given a touch of emphasis. Try using your imagination to select and use different characters to be printed in a border and observe the designs created.



1 Write a program that prints a single line border of X's around the edge of the screen.

2 Change your program so that the border will be two columns by two rows wide all around.

3 Revise the program on page 96 so that the screen border is made of the letter Q and the word QUESTION is printed at four different places on the screen.

4 How can a border be kept on the screen after a HOME statement is executed and a new frame is printed?

5 Program a routine that prints a frame with both a message and a screen border for an educational computer program you are now writing or plan to write.

3.8 Input Question Mark

An input statement such as INPUT P$ causes a question mark (?) to be printed on the screen. The question mark signals the user that a response should be typed, followed by a press of the Return key. In educational programs there are times when the input question mark will detract from the screen format. An input can be programmed without a question mark appearing on the screen by including a null string directly after the INPUT statement. A null string consists of a pair of quotation marks with no space in between. For example, 40 INPUT " ";P$. Note the null string that follows the INPUT statement.

In developing your program, you must decide, depending on the kind of input required, whether the INPUT statement should print a question mark. The next program prints a word on the screen and uses an input without a question mark. It calls for the user to pluralize the word cockroach by adding the correct ending. Absence of the question mark reduces screen clutter and allows the added ending to be printed in juxtaposition to the word.

```
5  HOME
10 W$ = "COCKROACH"
15  PRINT "ADD AN ENDING TO MAKE THE PLURAL FORM OF THE WORD. THEN PRES
    S RETURN."
20  VTAB 10
25  PRINT W$;
40  INPUT "";P$
50 PW$ = W$ + P$
60  IF PW$ = "COCKROACHES" THEN  PRINT "CORRECT!"
```

Here is an explanation of how the lines in the program are executed. Line 25 prints COCKROACH, the string assigned to W$ in Line 10. Line 40 programs an input without a question mark being printed on the screen. The semicolon after W$ causes the entry to print at the end of COCKROACH. If the correct plural ending is entered, the pluralized word appears on the screen as COCKROACHES. PW$ in Line 50 is assigned the concatenated string combination of COCKROACH (W$) and ES(P$). Line 60 checks for the correct plural form, COCKROACHES, and prints CORRECT if the

input pluralized the word correctly. Omitting the input question mark and using TABs to specify the location at which the input is printed can be useful in programs that involve arithmetic operations such as addition, multiplication, and subtraction. This short program illustrates this programming technique.

```
1  HOME
5  PRINT "MULTIPLY THE TWO NUMBERS."
7  PRINT "TYPE YOUR ANSWER AND PRESS RETURN."
8  VTAB 8
10  PRINT  TAB( 20)"3"
20  PRINT  TAB( 18)"X 2"
30  PRINT  TAB( 19)"---"
40  HTAB 20
50  INPUT "";N
60  IF N = 6 THEN  PRINT "CORRECT"
```

```
   3
 x 2
---
```

Figure 3.8
Multiplication problem.

When the program is run the problem appears on the screen as shown in Figure 3.8. Note the cursor positioned directly below the numbers. It is at this position the answer will be printed when typed as an input. Line 50 accepts a numeric input without printing a question mark and assigns the number to the variable N. If the correct answer, 6, is entered, Line 60 prints CORRECT.

ON LINE 3.8

1. What is an input question mark and when is it printed? What is the purpose of the question mark?

2. How would you use HTAB and VTAB to position the printing of an input question mark?

3. Write a short program that asks a question requiring a string input. Use VTAB, HTAB, and INPUT statements to print the entry in the center of the screen.

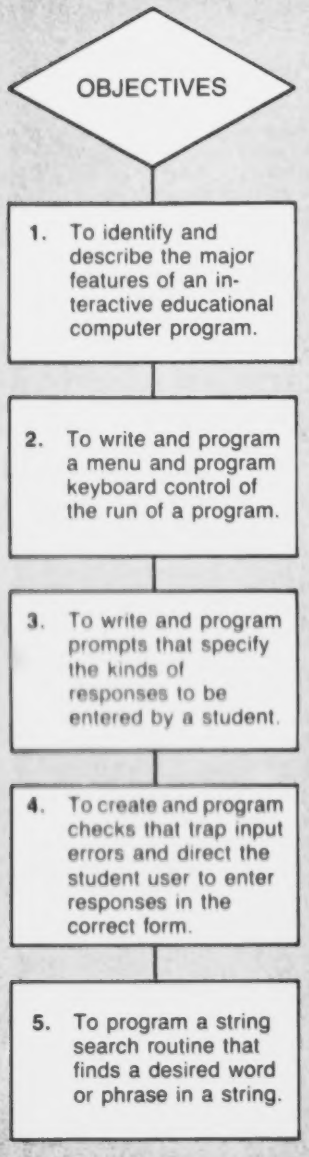
4. Rewrite the program so that the question mark for th input is not printed.

5 When might it be desirable to have an INPUT executed without the printing of a question mark?

6 Illustrate this technique by programming an example other than the two examples given in this section.

CHAPTER FOUR

INTERACTIVE


OBJECTIVES
1. To identify and describe the major features of an interactive educational computer program.
2. To write and program a menu and program keyboard control of the run of a program.
3. To write and program prompts that specify the kinds of responses to be entered by a student.
4. To create and program checks that trap input errors and direct the student user to enter responses in the correct form.
5. To program a string search routine that finds a desired word or phrase in a string.

This chapter discusses techniques for making a program interactive. For example, instruction is provided for writing and programming a menu, and programming keyboard control, questions, prompts, inputs, and feedback messages. You will learn to write routines that enable a student to use the keyboard to control the run of a program. A routine that directs a student to advance to the next frame by pressing the Space Bar is one kind of interactive feature covered here. Directions are given for writing prompts and questions that specify the kind of inputs required during a program run. Examples of different kinds of inputs common to educational programs are given.

The importance of assigning inputs to string variables and programming checks that catch undesirable inputs is explained. Directions are given for programming different kinds of input checks and providing feedback that directs the student's attention to reentering information correctly. A unique programming feature explained is a string search to locate a correct word or phrase in an input entered by a student in response to a question.

4.1 Interactive is Important

One of the most exciting, challenging, and useful characteristics of the computer is its interactive capability. For educational programs, interactive means that the student enters information in response to prompts from the computer and the computer provides feedback based on these inputs. Each, computer and student, has a reciprocal influence on the other.

While running an interactive program the student is definitely an active participant. This learning mode contrasts with the passive viewing of television or movies, or listening to a lecture. When a student is passively viewing or listening, it is easy for his mind to wander away from the subject or for him to miss some of the important points in the presentation. However, the computer-using student must be attentive to the questions and prompts presented on the video screen by an interactive program. When the student enters the requested inputs, the computer prints out feedback, such as VERY GOOD! or INCORRECT, that guides the student's learning. Because it directly commands the student's attention and involvement, the properly designed interactive educational program is a highly effective teaching device.

You can include interaction in your program in many ways. For example, if the program includes a tutorial, pretest, and quiz, you

should include a frame that presents a menu from which the student can select an option. A menu provides interaction by displaying choices and prompting the student to select that part of the program he or she wishes to run. Another important aspect of interaction is student control of the display of frames during the program run. This feature may be included in a program by printing PRESS SPACE BAR TO CONTINUE at the bottom of a frame and using the GET statement to control clearing the screen and subsequent printing of the next frame. Further, providing periodic opportunities to branch to different parts of the program also contributes to the interactive features.

Student inputs in response to prompts and questions are fundamental to the psychology of the computer learning experience. When the student reads a prompt or question, he must mentally formulate a response. This may involve memory recall, analysis, computation, or some other mental process. The response from the computer based on the student's input indicates whether the input is right or wrong and helps the student learn the material correctly.

Drill and practice programs exemplify the simplest kind of interaction. These programs may present a formula, rule, or series of rules for some operation. For example, converting Celsius temperature to Fahrenheit, use of *i, e,* and *c* in spelling, and classifying parts of speech. Drill and practice programs have value in learning things that require repetition for mastery of the material: for example, learning multiplication tables, learning dates of historical events, and balancing a chemical equation.

A more advanced type of tutorial program involves the student in a Socratic dialogue, an example of which follows.

```
Computer:  WHICH ANIMAL FEEDS ITS YOUNG ON MILK: HORSE,
           LIZARD, FISH?
Student:   HORSE
Computer:  CORRECT!
Computer:  DOES A HORSE HAVE HAIR ON ITS BODY?
Student:   YES
Computer:  RIGHT! A HORSE AND OTHER MAMMALS HAVE HAIR.
Computer:  IS A HORSE WARM BLOODED OR COLD BLOODED?
Student:   WARM BLOODED
Computer:  YES, ALL MAMMALS ARE WARM BLOODED.
Computer:  FACT: A HORSE IS A MAMMAL.
Computer:  NAME A CHARACTERISTIC OF A MAMMAL.
Student:   FEEDS ITS YOUNG ON MILK
Computer:  CORRECT!
Computer:  NAME ANOTHER CHARACTERISTIC OF A MAMMAL.
Student:   HAIR
Computer:  O.K. I SEE YOU KNOW THE CHARACTERISTICS OF MAMMALS.
Computer:  IS A LIZARD A MAMMAL?
Student:   NO
```

The student running this program continually interacts with the computer. Through the sequence of questions and responses, she learns the material. Creating a Socratic dialogue of this kind involves complex programming because provision must be made for different possible responses and appropriate feedback by the computer. This is the kind of program in which the string search routine (Section 4.7) can be used to search a student's free response for a key word or term that indicates a correct answer. For instance, the string search could be used to find the desired answer MILK, HAIR, or WARM BLOOD in the student's response to the prompt NAME A CHARACTERISTIC OF A MAMMAL. Dialogue programs illustrate the highest degree of interaction between the computer and student that is fundamental to the learning process.

```
950  REM  STUDENT'S SELECTION OF ANIMAL TO MATCH HEART BEATS
960  VTAB 4
970  PRINT "WHICH ANIMAL HAS A HEART THAT BEATS AT  THE RATE YOUR JUST H
     EARD?"
980  PRINT
990  PRINT  TAB( 15)"1. ";M$
1000  PRINT  TAB( 15)"2. ";C$
1010  PRINT  TAB( 15)"3. ";P$
1020  PRINT  TAB( 15)"4. ";E$
1040  PRINT
1050  PRINT "PRESS THE NUMBER OF THE CORRECT ANSWER. THEN PRESS RETURN."

1060  VTAB 20: HTAB 17
1070  INPUT A$
1080 A = VAL (A$)
1090  IF A < 1 OR A > 5 THEN  SPEED= 50: PRINT "PRESS AN NUMBER FROM 1 T
     HROUGH 5.  THEN PRESS RETURN.": SPEED= 255: GOTO 1060
1110  REM  CHECKS STUDENT'S ENTRY AS CORRECT OR INCORRECT
1120  HOME : VTAB (5)
1130  IF A = M(C) THEN  PRINT "CORRECT ANSWER!": GOTO 1150
1140  PRINT "SORRY, WRONG ANSWER."
```

Figure 4.1
Interaction occurs when the student responds to the question printed by Line 970. Note the feedback provided in Lines 1130 and 1140.

At an even more advanced level, simulation programs require student inputs that modify the real-world variables modeled in the simulation. One such program simulates the flight of an airplane

and portrays on the video screen the view seen by a pilot looking out the front window of the airplane's cockpit. As the program runs, the student presses different keys that regulate the controls of the simulated airplane and thereby change its speed and direction. The student observes on the screen the effect of the keyboard entries on the airplane's flight path. Visual feedback from the screen gives him the feeling that he is actually flying a real airplane.

If the airplane is not flown properly, the student eventually observes the cockpit view of the airplane flying down and into the ground. When this unfortunate event occurs, the screen darkens and CRASH! is printed, providing direct negative feedback in no uncertain terms. By repeatedly using the flight simulator program, the student learns to fly the airplane in takeoffs and landings at different airports under different atmospheric conditions. Advanced computer simulations of this type are now used extensively in training commercial airline and military pilots.

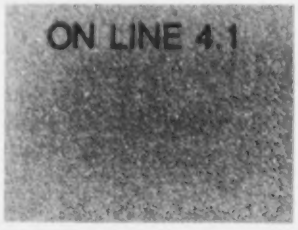
In summary, let it be said that an educational computer program, be it drill and practice, tutorial, or simulation, takes advantage of the computer's capabilities to the greatest extent when it is highly interactive. Therefore, in designing and developing your program, include as much interaction between the computer and student as possible so that when he is running the program he will be an involved participant. Avoid programming lengthy text (such as a glossary) for the student to read all at one time or repeating frames the student must study passively while these frames are displayed. When such material is a required part of the content, design the program so that the information is interspersed with questions or prompts. Or allow the user the option of branching to the frames containing this material and then returning to the main program. Proper attention to the interactive aspects of a program will contribute materially to its value and success.

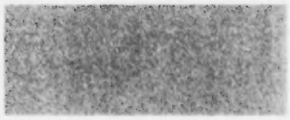
---

1 What is meant by interaction in an educational computer program?

2 Why is interaction a vital part of an educational computer program?

3 Write a short routine that creates interaction through use of the GET statement.





4 Write a routine that calls for an entry in response to a question and evaluates the entry as correct or incorrect. The routine should also provide feedback to the user after the entry is made.

4.2 User Control

User control is a feature of an educational program that gives the student user command of frame changes during the run of the program. Generally, once a frame is shown, passage to the next frame should depend on the user pressing a designated key to clear the screen and continue with the program. This key, commonly the Space Bar or the Return Key, is used consistently throughout the program. The following code forms a typical subroutine that allows the user to control the advance of the program from one frame to the next.

```
1000  VTAB 24: HTAB 4: PRINT "PRESS SPACE BAR TO CONTINUE ";
1010  GET K$
1020  IF K$ <  > " " THEN 1010
1030  HOME
1040  RETURN
```

The following line by line description of this subroutine illustrates the ease with which you can program user control.

1000 Prints a message at the bottom of the screen. Since the message is printed on the bottom screen line, it is imperative to end with a semicolon to prevent scrolling.
1010 Accepts a keyboard stroke and assigns it to the string variable K$. The GET statement works without the user pressing Return. Thus, once Line 1010 is executed, the program immediately proceeds to Line 1020.
1020 Returns to Line 1010 if the Space Bar is not the keystroke assigned to K$.
1030 Clears the screen if the Space Bar is pressed.
1040 Returns to the main part of the program to continue with the next frame.

If your educational program is menu driven, it is a good idea to allow the student user to return to the menu at any time. This feature enables exiting one section of a program and entering

another. It also frees the user from being forced to view an entire program even though he or she may already be familiar with the material being presented. The "returning to the menu" feature also allows the user to easily restart the program without viewing it in its entirety. This option is particularly important if the student missed key points in the earlier stages of the program. To add this feature to the previous subroutine, simply alter the message and insert a second conditional statement.

```
1000  VTAB 23: HTAB 4: PRINT "PRESS SPACE BAR TO CONTINUE"
1005  HTAB 9: PRINT "PRESS M FOR MENU ";
1010  GET K$
1015  IF K$ = "M" THEN  HOME : GOTO 2000
1020  IF K$ <  > " " THEN 1010
1030  HOME
1040  RETURN
2000  REM MENU
```

The adjustments made to the original user control subroutine are as follows:

1000 Message placed at Row 23 to create room for inserting the menu option statement. Also, the semicolon is deleted.
1005 Message indicating option of returning to menu.
1015 If the keystroke entered is an M, then the screen is cleared and program execution goes to Line 2000, which begins the print of the menu.

Although the adjustments to the subroutine are minimal, careful attention to the structure of the program is required when including the option of returning to a menu during the run of a program. Since the coding of a menu is generally near the beginning of a program, precaution must be used when positioning READ and DIM statements. If one chooses to return to the menu during the run, the program must be structured to avoid out of data errors and redimensioning errors. Coding the program to read all DATA statements and dimensioning all arrays before programming the menu is the surest way to avoid these errors. According to your specific requirements, however, you may find the need to read DATA statements and dimension arrays after the menu is run. If so, use the RESTORE statement to reset the data pointer to the first data element and the CLEAR statement to reset all array elements to zero. For more information about menus, refer to Section 4.9.

User control routines can be adjusted to provide information to the user. If the program includes a quiz, then the message depicted at the bottom of the screen can also inform the user of the number of the upcoming question. This next user control subroutine includes a counter that tracks the number of the question to be displayed on the screen. (See Section 5.2, Counters and Tallies.)

```
1000  GOSUB 2000
1990  END
2000 N = N + 1
2010  IF N > 5 THEN 3000
2020  VTAB 24: PRINT "PRESS Q FOR QUESTION # ";N;
2030  GET K$
2040  IF K$ <  > "Q" THEN 2030
2050  HOME
2060  RETURN
3000  VTAB 24: PRINT "PRESS S FOR SUMMARY ";
3010  GET K$
3020  IF K$ <  > "S" THEN 3010
3030  REM  SUMMARY ROUTINE FOLLOWS
```

Similar in style to the previous routines providing user control, this code informs the student user of the number of the upcoming question and can easily be adapted for any quiz section of a program.

2000 Increments the counter by 1.
2010 After the fifth question, program execution will shift to a summary routine that follows Line 3030.
2020 Prints a message at the bottom of the screen that includes the upcoming question number.
2030 Accepts a keyboard stroke and assigns it to the string variable K$.
2040 Returns to Line 2030 if Q is not pressed.
2050 Clears the screen if Q is pressed.
2060 Returns to the main program to display the next quiz question.
3000 Prints the prompt indicated after the fifth question is answered.
3030 Beginning of routine that will provide a summary of the student's work.

Similar user control routines can be created by making minor adjustments to the codings listed. For example, you may alter the directions or direct that a key other than any of those used before be pressed for the program to continue. Overall, however, user

control routines are standard and are an expected feature of educational programs.

```
1430 REM  QUIT OPTION
1440 VTAB 24: HTAB 10: PRINT "TRY AGAIN (Y/N)?";
1450 GET K$
1460 IF K$ = "Y" THEN 750
1470 IF K$ < > "N" THEN 1450
1480 TEXT : HOME : END
```

Figure 4.2
Quit option in The Number Machine. Line 1440 prints TRY AGAIN (Y/N)? If the student presses Y, the program goes to Line 750 for another entry. If N is pressed, the program proceeds to Line 1480 and ends. Line 1470 checks for an entry other than N, control returning to Line 1450 if the condition K$< > "N" is true.

---

1 What two keys are generally used to program user control routines? Why?

2 Alter the user control routine on page 106 to accept any keyboard entry to continue.

3 List two advantages of allowing the user to press any key to continue the run of the program.

4 List two disadvantages of allowing the user to press any key to continue the run of the program.

5 Write a user control subroutine that prints a message incorporating the user's name each time the routine is called.

Prompts 4.3

You can use prompts in your educational computer program to guide students to correct responses. A prompt is a hint or cue that provides information the student can use to arrive at the correct response. One example of the use of prompts is a program designed

to help students learn the names of state capitals. Here is a set of five BASIC statements that are a part of this program.

```
1. PRINT "THE CAPITAL OF CALIFORNIA IS          ."
2. PRINT "THE CAPITAL OF CALIFORNIA IS ----------."
3. PRINT "THE CAPITAL OF CALIFORNIA IS S---------."
4. PRINT "THE CAPITAL OF CALIFORNIA IS SA--------."
5. PRINT "THE CAPITAL OF CALIFORNIA IS SAC--MEN-O."
```

Examine these statements carefully. There is no prompt in the first statement. This is called a criterion or terminal statement: the student either knows or does not know the correct answer. Each of the next four statements contains a prompt that provides cues to the student.

Statement 1 is presented to the student first. If the student knows the correct response is SACRAMENTO and answers correctly, the program moves on to questions for other states. However, if the student answers incorrectly, the question is repeated in the form of statement 2, in which the number of dashes indicates the number of letters in the name of the capital. If this question is missed, statement 3, which supplies the number of letters in the name of the capital and the first letter of the name, is presented. If necessary, additional cues are provided in statements 4 and 5.

If the student is incorrect in answering the question at statement 5, then the correct answer is printed and the program proceeds to other states and capitals, providing similar kinds of prompts. On repeated runs of States and Capitals the student proceeds through the program more rapidly because he or she is learning the names of the capitals and is more likely to get the correct answer on the first try. The order in which the states are presented may be randomized to provide variety in use of the program.

One way to program the set of statements for the capital of California is as follows:

```
90 S$ = "SACRAMENTO"
95 F = 0
100  PRINT "THE CAPITAL OF CALIFORNIA IS                    ."
110  INPUT A$
120  GOSUB 5000
125  IF F = 1 THEN 250
126  PRINT "THE CAPITAL OF CALIFORNIA IS - - - - - - - - - -."
127  INPUT A$
128  GOSUB 5000
129  IF F = 1 THEN 250
130  PRINT "THE CAPITAL OF CALIFORNIA IS S - - - - - - - - -."
140  INPUT A$
```

```
150  GOSUB 5000
155  IF F = 1 THEN 250
160  PRINT "THE CAPITAL OF CALIFORNIA IS S A - - - - - - - - O."
165  INPUT A$
170  GOSUB 5000
175  IF F = 1 THEN 250
190  PRINT "THE CAPITAL OF CALIFORNIA IS S A C - - M E N - O."
195  INPUT A$
200  GOSUB 5000
210  IF F = 1 THEN 250
220  PRINT "SACRAMENTO IS THE CAPITAL OF CALIFORNIA."
250 F = 0: PRINT "PROGRAM CONTINUES FROM THIS POINT"
4999  END
5000  IF A$ = S$ THEN  PRINT "CORRECT!":F = 1: GOTO 5020
5010  PRINT "SORRY,WRONG ANSWER!"
5020  RETURN
```

S$ contains the correct name for the capital of California. F is initialized to 0 at Line 95. After the student types the name of a capital at Line 110, the program branches to the short subroutine beginning at Line 5000. If the name is correct, the message CORRECT! is printed and F is incremented to 1. The program branches to Line 5020, which branches to Line 125. Since F = 1, the program branches to Line 250, at which point the program continues and prints the name of another state not shown in our example.

If the entered name is incorrect, the message SORRY, WRONG ANSWER! is printed, and the program branches to Line 5020 and to Line 125. Since F is not equal to 1, the program drops through to Line 126, where the name of the same capital is again requested, but this time a prompt is also provided to help the student arrive at the correct answer. At each call to the subroutine, F either remains 0 or is incremented to 1. If the student continues to type an incorrect answer, additional statements at Lines 130, 160, and 190, which contain increasingly helpful hints, are executed. However, if the student does not get the correct answer in the fifth try, the computer prints the name of the capital and then proceeds to ask for the capitals of other states.

It is important to understand that the purpose of a prompt is to limit the range of possible responses without revealing the correct answer. The preceding prompts are referred to as formal prompts. Formal prompts provide only information concerning the form of the correct response, such as the position and name of the first letter, and the number of letters.

Overuse of formal prompts in educational programs should be avoided because students can sometimes provide a correct response, based on a strong formal prompt, and not really understand why

the response is correct. Hence, it may be necessary to emphasize thematic prompts. Thematic prompts encourage students to use past associations of words, phrases, or concepts to arrive at the correct response. As contrasted with formal prompts, thematic prompts inform the student about the correct response without revealing the form of that response. They provide greater evidence that a student has responded in terms of the meaning or theme of the subject matter being presented. Here are two statements in BASIC that include thematic prompts.

```
10 PRINT "SACRAMENTO IS THE CAPITAL OF THE 'GOLDEN' STATE. SACRAMENTO I
   S THE CAPITAL OF                    ."
20 PRINT "FRANKFORT IS THE CAPITAL OF THE 'BLUE GRASS' STATE. FRANKFORT
    IS THE CAPITAL OF                  ."
```

The association suggested by each of these statements is quite strong if the student possesses the requisite background. If not, then a combination of formal and thematic prompts may be particularly effective. Here are two statements in BASIC that include both formal and thematic prompts.

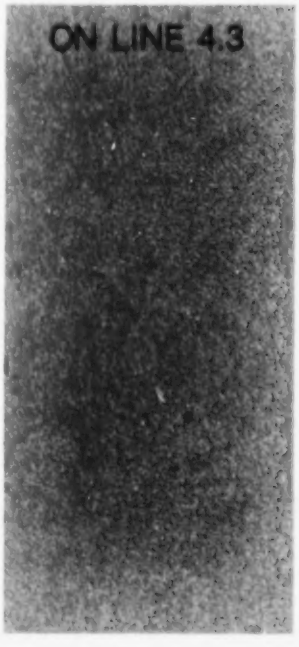
```
10 PRINT "BOISE IS THE CAPITAL OF THE 'POTATO' STATE. BOISE IS THE CAPI
   TAL OF I       ."
20 PRINT "AUSTIN IS THE CAPITAL OF THE 'LONGHORN' STATE. AUSTIN IS THE
   CAPITAL OF T       ."
```

The following program is an example of how you can use both formal and thematic prompts in the same educational computer program. It is intended to help students associate the names of capitals with their respective states.

```
100  PRINT "SACRAMENTO IS THE CAPITAL OF                    ."
110  INPUT S$
115  REM  PROGRAM LINES OMITTED
200  PRINT "SACRAMENTO IS THE CAPITAL OF THE 'GOLDEN' STATE."
210  PRINT "SACRAMENTO IS THE CAPITAL OF                    ."
220  INPUT S$
225  REM  PROGRAM LINES OMITTED
300  PRINT "SACRAMENTO IS THE CAPITAL OF A WEST COAST STATE."
310  PRINT "SACRAMENTO IS THE CAPITAL OF                    ."
320  INPUT S$
325  REM  PROGRAM LINES OMITTED
400  PRINT "SACRAMENTO IS THE CAPITAL OF C - - - - - - - - -."
410  INPUT S$
415  REM  PROGRAM LINES OMITTED
```

```
500  PRINT "SACRAMENTO IS THE CAPITAL OF C A - I F - - N - A."
510  INPUT S$
```

Inspecting these statements shows that, initially, there is no prompting, but then prompting moves from thematic to formal. The intent is to provide the student with every opportunity to come up with the correct state before the program gives the answer. If the student gets it right without prompting, the program moves on to present another capital and state. However, if the student needs help, the prompts give hints to the correct answer. If the student makes an incorrect entry at Line 110, Line 200 prints the first prompt, 'GOLDEN' STATE. Subsequent prompts are given by Lines 300, 400, and 500 if the student continues to enter the incorrect answer. Five opportunities are provided to enter an answer, and four of these opportunities provide a prompt that cues the correct answer. As the student uses the program for all 50 states repeatedly, he or she eventually learns the capitals and will be able to enter the correct answers on the first try.



1 Why are prompts used in an educational computer program?

2 Distinguish between formal and thematic prompts.

3 Thematic prompts are more effective than formal prompts in eliciting knowledge relevant to the subject being presented. Explain.

4 Write three formal prompts appropriate for inclusion in a program designed to provide instruction on the U.S. system of coinage.

5 Write three thematic prompts for question 4.

6 Why is it necessary to decrease prompting as the student progresses through an educational computer program?

7 Write a prompt that is both formal and thematic.

4.4 Inputs

An educational computer program contains questions and input statements that require the student to enter information during a run of the program. Good classroom teaching is often associated with questions that stimulate student thinking and enable the teacher to assess how well students are learning. The same principle is true of an interesting and interactive program. Through inputs the student has an opportunity to interact with the content being presented by the program. Moreover, the student receives feedback based on his or her entries that contributes to the student's learning.

You are likely to be familiar with the use of numeric variables such as N or X and string variables such as N$, A$, and NR$ in input statements. Although it is possible to use numeric inputs for numbers, we recommend you write your programs so that all inputs are accepted as string variables. A case in point: suppose a student is asked to enter his age by entering the number. He enters 18, and the input is assigned to a string variable, A$ = "18". The coding for this example is

```
100  PRINT "PLEASE TYPE YOUR AGE.  THEN PRESS        RETURN."
110  INPUT AG$
```

There is a definite advantage to accepting all inputs, including numbers, as strings. For instance, when a number is accepted as an input and assigned to a string variable, you can use string functions (LEFT$, RIGHT$, MID$) to locate specific digits in the string. One example of this technique is given in the next routine. A locker number is accepted as an input and assigned to the string variable L$. The first digit in the number is used to assign the student to a group.

```
10  PRINT "ENTER THE NUMBER OF YOUR LOCKER."
20  INPUT L$
30  PRINT "YOU ARE ASSIGNED TO GROUP "; LEFT$ (L$,1)

]RUN
ENTER THE NUMBER OF YOUR LOCKER.
?234
YOU ARE ASSIGNED TO GROUP 2
```

Suppose the student enters 234 as the locker number. Line 30 prints the first digit in the number, as identified by the LEFT$

function. Since LEFT$(L$,1) = "2", YOU ARE ASSIGNED TO GROUP 2 is printed on the video screen.

If you accept numeric inputs as string variables, you can also use the numeric value of the string variable in calculations. This is done with the VAL function, which assigns the numeric value of a string variable to a numeric variable. For example, A$ = "18". The statement A = VAL (A$) will assign the numeric value of 18 to A.

The following program lines illustrate the technique of accepting an input number as a string variable and assigning the numeric value of the string to a numeric variable.

```
100  PRINT "TYPE THE YEAR YOU WERE BORN? THEN PRESS RETURN."
110  INPUT Y$
120 Y =  VAL (Y$)
130  PRINT "IN THE YEAR 2000 YOU WILL BE ";2000 - Y;" YEARS   OLD."
140  PRINT "THE LAST TWO DIGITS IN THE YEAR OF YOUR BIRTH ARE "; RIGHT$
    (Y$,2)
150  PRINT "WRITE THIS NUMBER DOWN.  IT IS YOUR     SECRET CODE FOR USE
    IN RUNNING THIS     PROGRAM."

]RUN
TYPE THE YEAR YOU WERE BORN? THEN PRESS RETURN.
?1945
IN THE YEAR 2000 YOU WILL BE 55 YEARS   OLD.
THE LAST TWO DIGITS IN THE YEAR OF YOUR BIRTH ARE 45
WRITE THIS NUMBER DOWN.  IT IS YOUR     SECRET CODE FOR USE IN RUNNING THIS
PROGRAM.
```

Line 110 accepts the year of birth as an input and assigns the string value to Y$. The numeric value of Y$ is assigned to Y in Line 120. This value is used in Line 130 to compute the student's age in the year 2000 by subtracting Y from 2000. In Line 140 the last two digits in the birth year are printed through use of the RIGHT$ function.

As explained in Section 4.5, accepting all inputs as string variables is also extremely useful in checking the appropriateness of inputs so that the run of a program will proceed smoothly.

---

1 Why is it recommended that all inputs be accepted as strings?

2 Explain the difference between AG$ = "10" and AG = 10.

3 Which of the two variables in question 2 can be used to calculate 10.3? State the reason for your answer.

4 Write a short program that requires a student to enter the name and date of a historical event. The program should accept all inputs as string variables as well as calculate and print the number of years ago that the event occurred.

4.5 Input Checks

Input checks should be included in the final form of your program to prevent it from "crashing" due to inputs that are not acceptable in the run of the program. What kinds of inputs can cause this kind of problem? Suppose a frame in your program directs a student to type a 0 or 1, and he types the letter *O* for a 0 or the letter *L* for a 1. Suppose the student is directed to type a number from 1 through 5 to select a choice out of five categories and he errs by entering a 10. Obviously, the inputted 10 cannot be used to select a category because only five are available. When an input error occurs, the program must print a message on the video screen that indicates to the user the nature of the error and the way to correct it. That's user friendly.

Here are the lines from a program in which the user is directed to enter a 0 or 1.

```
190 VTAB 10
200 PRINT "TYPE 0 OR 1.  THEN PRESS RETURN."
210 INPUT N$
220 IF (N$ <  > "0") AND (N$ <  > "1") THEN  SPEED= 10: PRINT "PLEASE T
   YPE 0 OR 1.": VTAB 12: CALL  - 868: SPEED= 255: GOTO 190
```

Any input other than 0 or 1 in Line 210 is detected or "trapped" by the condition in Line 220. Line 220 is executed if the string value of N$ is less than "0" or the string value is greater than "1".

The error message PLEASE TYPE 0 OR 1 is printed by Line 220. VTAB 12, the next statement after the error message, moves the cursor to the beginning of the line on which the message is printed. CALL -868 in Line 220 clears the message from the screen, leaving a blank line on which the message can be printed again if the student inadvertently makes another entry other than 0 or 1. Control then goes to Line 190, allowing the student user to enter an acceptable

input at Line 210. If a 0 or 1 is entered, Line 220 is not executed and the program continues.

When Line 210 is executed, the student's entry is assigned to N$. However, if the first input is unacceptable, the second input is assigned as the new value of N$. For instance, if the student's first input was an *L*, the value of N$ would be "L" (N$ = "L"). After the student reads the error message and types a second input, 1, the execution of Line 210 changes the value of N$ from "L" to "1" (N$ = "1").

This listing shows the trapping of numbers outside of an acceptable range of 1 through 5.

```
290  VTAB 20
300  PRINT "CHOOSE A CATEGORY.  TYPE THE NUMBER OF  YOUR CHOICE."
310  INPUT N
320  IF N < 1 OR N > 5 THEN  SPEED= 25: PRINT "TYPE A NUMBER FROM 1 THRO
     UGH 5.": VTAB 23: CALL  - 868: SPEED= 255: GOTO 290
```

The program will accept only numbers 1 through 5, as indicated by the conditional statement in Line 320. If N < 1 or N > 5, then the speed of printing is set at the slower rate of 25 (SPEED = 25), the error message is printed, and the student is given an opportunity for a second input. Slowing the speed at which the error message is printed calls attention to the message. Note that the speed is reset to the normal rate of 255 after the error message is printed. Otherwise, the program would continue to run at the slower rate, an annoyance no user would appreciate.

Figure 4.3
The Number Machine checks the input of the student's grade level. A message is printed indicating an unacceptable entry if the entered grade is outside the range of 1 through 12.

Sometimes an input error occurs when a student inadvertently presses the Return key before typing the input. If the input variable is numeric, the computer prints a ?REENTER message. If the input variable is a string variable, a null string ("") is assigned as the value of the string variable. Essentially, the variable has the value of "nothing." The following program shows how this potential input error may be trapped by accepting the input, numeric or alphabetic, as a string.

```
100 VTAB 10
110 PRINT "PLEASE TYPE YOUR NAME."
115 PRINT "THEN PRESS RETURN."
120 INPUT N$
130 IF N$ = "" THEN  SPEED= 50: PRINT "YOU PRESSED THE RETURN KEY FIRST
    .  TYPE YOUR NAME.  THEN PRESS RETURN.": VTAB 13: CALL  - 868: VTAB
    14: CALL  - 868: SPEED= 255: GOTO 100
```

Line 130 checks for a null string. If one is detected, an error message is printed, and the program accepts another entry (Line 120) after printing the directions specified in the prompts (Lines 110 and 115).

An input error may occur in the use of a multiple choice question in which the student must input a letter to indicate the answer. For example, HOW MANY BITS EQUAL ONE BYTE? A. FOUR B. EIGHT C. SIXTEEN. Here the student should type the correct answer, B, and press Return. However, the student may mistakenly type the answer EIGHT instead of the letter that identifies the answer. Or the student may accidentally enter a letter other than A, B, or C, which are the only acceptable choices. How can these kinds of errors be trapped? One solution to the problem is as follows:

```
500 VTAB 10
510 PRINT "HOW MANY BITS EQUAL ONE BYTE?"
515 PRINT
520 PRINT  TAB( 4)"A. FOUR  B. EIGHT  C. SIXTEEN
522 VTAB 20: PRINT "TYPE A, B, OR C.  THEN PRESS RETURN."
525 VTAB 15: HTAB 19
530 INPUT AN$
535 PRINT
537 IF  LEN (AN$) > 1 THEN  SPEED= 50: PRINT "TYPE A, B, OR C.  THEN PR
   ESS RETURN": SPEED= 255: HOME : GOTO 500
540 IF AN$ < "A" OR AN$ > "C" THEN  SPEED= 50: PRINT "TYPE THE LETTER A
     B  OR  C TO SELECT AN ANSWER.THEN PRESS RETURN.": SPEED= 255: HOME
   : GOTO 500
550 HOME

    ]RUN
    HOW MANY BITS EQUAL ONE BYTE?
```

```
   A. FOUR  B. EIGHT  C. SIXTEEN
TYPE A, B, OR C.  THEN PRESS RETURN.
                  ?FOUR

TYPE A, B, OR C.  THEN PRESS RETURN
```

The question in Line 510 is printed at Row 10 as indicated by VTAB 10 in Line 500. Line 525 sets the printing of the input question mark at Row 15, Column 19, centered below the question. If an entry other than A, B, or C is made, the error is trapped in Line 540. The error message in Line 540 is printed, and the program goes back to Line 500, allowing for a second, correct entry. When the appropriate input is made, Line 550 clears the screen in preparation for the next frame.

Do not overlook the use of subroutines when programming input checks. Subroutines should be used for those routines that are repeated during the run of a program. For example, if the program involves a multiple choice quiz, you can use the input checks in the preceding program as a subroutine. After each input statement, use a GOSUB that calls the input check subroutine.

You may find it advisable to include in your program a subroutine that checks the characters in a string input for acceptability. For this purpose a FOR/NEXT loop is advisable. In the next example the student must input the year of a historical event (for example, 1492). The program calls the subroutine, which checks each character in the string to be certain it contains only numbers.

```
600  PRINT "TYPE THE YEAR OF THE EVENT.  THEN PRESS RETURN."
620  INPUT Y$
630  GOSUB 5000
635  IF F = 1 THEN F = 0: PRINT "TYPE ONLY NUMBERS.": PRINT : GOTO 600
4999  END
5000  FOR I = 1 TO  LEN (Y$)
5010  IF  MID$ (Y$,I,1) < "0" OR  MID$ (Y$,I,1) > "9" THEN F = 1
5020  NEXT I
5030  RETURN

]RUN
TYPE THE YEAR OF THE EVENT.  THEN PRESS RETURN.
?L930
TYPE ONLY NUMBERS.

TYPE THE YEAR OF THE EVENT.  THEN PRESS RETURN.
?
```

In response to the prompt in Line 600 the student enters a year, and the string value of the entry is assigned to Y$ by Line 620. Line

630 calls the subroutine (Lines 5000-5030), which contains a FOR/NEXT loop that checks Y$ for characters other than numbers. Each character in Y$ is checked in succession by Line 5010 as the loop runs. If a character (Y$, I, 1) is outside the string value range from "0" to "9", the flag F is assigned a value of 1. If F = 1 when the program returns to Line 635, the value of F is set to 0 and the error message TYPE ONLY NUMBERS is printed. Control of the program branches to Line 600 for another printing of the directions and acceptance of an input by Line 620.

Consider the following two lines in which an entry is requested and assigned to a string variable:

```
100  PRINT "ENTER YOUR FIRST NAME."
110  INPUT N$
```

Sometimes a student unconsciously presses the Space Bar after entering a response to a question like that in Line 100. When this occurs, the space is included in the string assigned to the string variable. If a student types LUIS and presses the Space Bar before pressing the Return key the string assigned to N$ would contain LUIS and a space after the S: "LUIS ". One or more unwanted spaces such as this can prove troublesome when printing a variable or concatenating strings. To solve the problem, use an input check such as the following to parse one or more trailing spaces at the end of a string.

```
100 PRINT "ENTER YOUR FIRST NAME."
110 INPUT N$
120 IF  RIGHT$ (N$,1) = " " THEN N$ =  LEFT$ (N$, LEN (N$) - 1): GOTO 12
    0
```

If RIGHT$(N$,1) is a space, the THEN statement assigns the string, without the space, to N$. The GOTO 120 sends control back to the IF statement in the line to check for another space. If there are no trailing spaces at the end of N$, the program goes to the next line.

The foregoing examples should help you understand and design input checks. After you have completed your educational computer program, evaluate each input in terms of likely input errors and then add the lines of code, either within the main program or as a subroutine, needed to trap these errors and thereby help to insure a smooth run of the program.

1 How can you trap null string entries?

2 Write lines of BASIC to show how a null string check may be programmed as a subroutine.

3 Write a routine that requires the entry of a number from 10 to 100 and assigns it to a string variable. Include a check that traps a number outside this range and requests the user to input an acceptable value.

4 Write a short program in which the student is requested to enter the trade name of a car that contains more than seven letters. Add an input check that traps unacceptable entries and requests the student to reenter a name if the first entry is unacceptable.

5 Write a short program that directs the student user to enter the name of a month that contains three letter *E*'s. Add an input check that uses the MID$ function to ascertain the presence of three *E*'s and provides feedback to the student as to whether the entry is correct.

Multiple Choice GET 4.6

Many educational computer programs include multiple choice questions the student must answer. For example:

```
HOW MANY CHAMBERS DOES A FROG'S HEART CONTAIN?
A. ONE  B. TWO  C. THREE  D. FOUR
```

In response to this question the student should press the letter of the correct answer, which is C. If the response is accepted as an input, the student must follow a press of the C key with a press of the Return key.

A more efficient technique for programming the response to a multiple choice question is to use the GET statement to accept the student's entry. GET assigns a single keystroke to a variable. For example, GET AN$ assigns to AN$ the character of the key that is pressed (e.g., AN$ = "C"). Unlike INPUT, a question mark does

not appear on the screen when GET is executed, and the character of the key pressed by the student does not appear on the screen. Also, with GET the student does not press the Return key when entering a response. This feature, therefore, enables the student to respond to a multiple choice question through a single keystroke and eliminates cluttering the screen with a superfluous question mark.

Following is an illustration of the programming of GET in accepting a student's response to a multiple choice question:

```
90  VTAB 10
100  PRINT "HOW MANY CHAMBERS DOES A FROG'S HEART   CONTAIN?"
105  PRINT
110  PRINT "A. ONE  B. TWO  C. THREE  D. FOUR"
120  VTAB 20
130  PRINT "PRESS THE LETTER FOR YOUR ANSWER."
140  GET AN$
160  IF AN$ = "C" THEN  PRINT "CORRECT!": GOTO 180
170  PRINT "SORRY, WRONG ANSWER."
180  FOR P = 1 TO 1000: NEXT P
190  HOME
```

The letter of the key pressed by the student is assigned to AN$. If C is pressed, AN$ = "C" and Line 160 prints CORRECT!. The next statement in Line 160, GOTO 180, sends the program to Line 180. After the delay (Line 180) Line 190 clears the screen, and the program moves on to printing the next frame. If the A, B, or D keys are pressed, the condition in Line 160 is false, and the program goes to Line 170, which prints SORRY, WRONG ANSWER.

The routine is enhanced by adding a check for an inadvertent press of the Return key, as shown in the added Line 150:

```
150  IF AN$ = "" THEN  PRINT "PRESS THE LETTER FOR YOUR CHOICE.  DO NOT
     PRESS RETURN.": GOTO 140
```

Further, it is desirable to "lock out" any keys other than A, B, C, and D. Lock out means that the run of the program will not continue if a key other than A, B, C, or D is pressed. If, for example, the student presses the F key, the frame with the question remains on the screen and the computer waits for a press of a key within the acceptable range, A through D. This feature may be included in the program by adding the line:

```
155 IF AN$ < "A" OR AN$ > "D" THEN PRINT "PRESS A,  B,  C  OR D.": GOTO
    140
```

Line 155 evaluates AN$ in terms of its ASCII value. If the value is less than "A" or greater than "D", a key other than A, B, C, or D has been pressed, causing the ASCII value of AN$ to be outside the range specified in Line 155. This evaluation results in the printing of the message PRESS A, B, C, OR D, and the program returning to Line 140 to accept another keystroke.

---

1 How is a GET statement similar to and different from an INPUT statement?

2 Why is it more advantageous to use GET rather than INPUT to accept a student's response to a multiple choice question?

3 Why should a program include a check for a GET statement assignment?

4 Explain how a GET statement can be used to "lock out" keys.

5 Write a subroutine to be used in a program to check a GET assignment for the response to each multiple choice question included in the program. The acceptable responses to each question range from A through D.

String Search 4.7

You can use a string search routine to find a desired word or phrase in a string input. Here is a question from a computer program that illustrates a string search: WHAT IS THE NAME OF THE FIRST PRESIDENT OF THE UNITED STATES? In responding to this question, the student should type WASHINGTON and press the Return key. However, a student might type GEORGE, GEORGE WASHINGTON, or even PRESIDENT WASHINGTON I BELIEVE. In these cases the input may be checked with a string search for the acceptable answer WASHINGTON.

The following routine searches the input string stored in PR$ for the substring WASHINGTON stored in N$. The actual search is accomplished by Lines 130–150.

```
100  PRINT "WHAT IS THE NAME OF THE FIRST PRESIDENT OF THE UNITED STATE
     S?"
110 N$ = "WASHINGTON"
120  INPUT PR$
130  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
140  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 180
150  NEXT I
160  PRINT "SORRY, BUT YOUR ANSWER IS NOT CORRECT."
170  GOTO 100
180  PRINT "YOUR ANSWER IS CORRECT."
```

Let's examine each line in the program to determine its purpose.

100 Student is prompted to type the name of a president.
110 The correct answer is stored in N$.
120 Student's answer is stored in PR$.
130 FOR/NEXT loop, based on values of LEN(PR$) and LEN(N$), determines the number of iterations (repetitions) necessary to search the string stored in PR$ for the string stored in N$.
140 Performs the actual search of the string stored in PR$ to determine if it contains the string stored in N$.
150 Increments I in the FOR/NEXT loop by 1.
160 Prints an appropriate message if no match is found.
170 Branches student to appropriate line for another attempt.
180 Prints an appropriate message if a match is found.

Now let's take a closer look at Lines 130 and 140, the real workhorses in the search strategy. The simplest case is that in which the student types the correct answer and nothing else. In this case, PR$ and N$ both contain the string WASHINGTON, so LEN(PR$) − LEN(N$) + 1 evaluates as 10 − 10 + 1 to give 1. Thus Line 130 is FOR I = 1 to 1, which indicates that only one iteration of the loop is required to determine if the string in N$ is contained in the string in PR$. Is one iteration sufficient? In Line 140 if we replace PR$ with WASHINGTON, I with 1, LEN(N$) with 10, and N$ with WASHINGTON, we have

```
140  IF  MID$ ("WASHINGTON",1,10) = "WASHINGTON" THEN 180
```

The comparison is

PR$ ⟶ WASHINGTON
N$ ⟶ WASHINGTON

There is, indeed, a match.

Here is a second example. The student types GEORGE WASHINGTON. Will the search routine work appropriately and locate the correct answer, WASHINGTON? Remember that a space (there is a space between GEORGE and WASHINGTON) is a character in BASIC and must be included in the search. In Line 130, LEN(PR$) – LEN(N$) + 1 evaluates as 17 – 10 + 1 to give 8. Thus our loop in this example is FOR I = 1 TO 8. Note that the value of I in Line 140 is incremented by 1 in each iteration of the loop. The incrementation successively moves the string stored in N$, WASHINGTON, one character to the right on the string stored in PR$, GEORGE WASHINGTON, before a comparison is made.

LOOP	COMPARISON
1	GEORGE WASHINGTON WASHINGTON
2	GEORGE WASHINGTON WASHINGTON
3	GEORGE WASHINGTON WASHINGTON
4	GEORGE WASHINGTON WASHINGTON
.	
.	
.	
8	GEORGE WASHINGTON WASHINGTON

A match is found in the eighth and final iteration of the loop. Again Line 130 correctly determined the maximum number of iterations required for the search.

Here is a third example. Suppose the student types PRESIDENT WASHINGTON I BELIEVE. In Line 130, LENL(PR$) – LEN(N$) + 1 evaluates as 30 – 10 + 1 to give 21. A total of 21 iterations of the loop is required if the correct answer is located at the end of the string stored in PR$. However, since WASHINGTON is not located at the end of the string in our third example only 11 iterations are required. Convince yourself that 11 iterations are sufficiently by comparing the strings, character by character, as we did in the second example.

Up to this point in our examples of a string search, we have varied only the number of characters in the string being searched, not in the substring for which we are searching. Would a change

in the number of characters in the substring affect the search routine? The answer is definitely no. The LEN(N$) statements in Lines 130 and 140 insure that the appropriate number of iterations in the FOR/NEXT loop will be used and that the appropriate number of characters constituting the substring will be compared to the input string in each successive iteration. For example, if the input string is ABE LINCOLN and the correct answer is LINCOLN, the search routine will locate the string LINCOLN as described in the previous examples. Convince yourself of this by playing computer and finding the string.

What happens if the substring N$ is longer than the input string PR$? Nothing spectacular. For example, if PR$ is WASH and N$ is WASHINGTON, LEN(PR$) − LEN(N$) + 1 evaluates as 4 − 10 + 1 to give −5. In Applesoft BASIC the loop would iterate one time. Then the program would execute the line following the NEXT statement, indicating that no match was found, which is correct. Again the search routine operates correctly.

Note that on some systems the comparison of a lowercase letter to its uppercase partner, such as H with h, would not give a match. That is WASHINGTON would not match Washington. So make certain the program informs the student to use uppercase, lowercase, or a combination of these as required when typing answers.

It is easy to include the string search routine in your program. Simply place the routine after each statement that requires an input to be searched for the correct answer. For example, your program on presidents of the United States might be as follows:

```
100  PRINT "NAME THE PRESIDENT, OTHER THAN FORD,WHO WAS NEVER ELECTED T
    O OFFICE."
110 N$ = "FILLMORE"
120  INPUT PR$
130  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
140  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 170
150  NEXT I
160  PRINT "SORRY, BUT YOUR ANSWER IS INCORRECT": GOTO 120
170  PRINT "YOUR ANSWER IS CORRECT."
180  PRINT "NAME THE FIRST PRESIDENT TO DIE IN OFFICE"
190 N$ = "HARRISON"
200  INPUT PR$
210  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
220  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 260
230  NEXT I
240  PRINT "TRY AGAIN": GOTO 200
260  PRINT "RIGHT ON"
270  PRINT "NAME THE ONLY PRESIDENT EVER TO BE IMPEACHED"
280 N$ = "JOHNSON"
```

```
285  INPUT PR$
290  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
300  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 340
310  NEXT I
320  PRINT "WHOOPS, THAT'S NOT IT": GOTO 285
340  PRINT "YOU GOT IT"
350  PRINT "NAME THE PRESIDENT WHO WON A PULITZER PRIZE FOR HIS BOOK,'P
    ROFILES IN COURAGE'."
355 N$ = "KENNEDY"
360  INPUT PR$
370  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
380  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 400
390  PRINT "YOU MISSED THAT ONE": GOTO 360
400  PRINT "VERY GOOD"
410  PRINT "THAT'S ALL FOR NOW."
420  END
```

We included the string search after each of the four questions about presidents. If the program had contained 20 such questions we would have had to, using this approach, type the search routine 20 different times. And what if the search routine contained 15 lines rather than only three? Well we would be typing for a long time to say the least. Clearly such a procedure is inefficient. If you plan to use a routine such as the search routine in your program more than a few times then you should place it in a subroutine. The subroutine can be called from the main program anytime it is needed to search an input string for the correct answer. Here is the previous program rewritten to include a subroutine.

```
100  PRINT "NAME THE PRESIDENT, OTHER THAN FORD, WHO WAS NEVER ELECTED T
    O OFFICE."
110 N$ = "FILLMORE"
120  GOSUB 900
160  PRINT "NAME THE FIRST PRESIDENT TO DIE IN OFFICE."
170 N$ = "HARRISON"
190  GOSUB 900
220  PRINT "NAME THE ONLY PRESIDENT TO EVER BE IMPEACHED."
230 N$ = "JOHNSON"
250  GOSUB 900
280  PRINT "NAME THE PRESIDENT WHO WON A PULITZER PRIZE FOR HIS BOOK, 'P
    ROFILES IN COURAGE'."
290 N$ = "KENNEDY"
310  GOSUB 900
340  PRINT "THAT'S ALL FOR NOW."
345  END
900  INPUT PR$
1000  FOR I = 1 TO  LEN (PR$) -  LEN (N$) + 1
1010  IF  MID$ (PR$,I, LEN (N$)) = N$ THEN 1050
```

```
1020  NEXT I
1030  PRINT "INCORRECT, PLEASE TRY AGAIN."
1040  GOTO 900
1050  PRINT "CORRECT"
1060  RETURN
```

The search works exactly as it did in the program with no subroutine. When the main program branches to the subroutine beginning at Line 900, the student's answer is requested. Then the input string is searched for the correct answer. If the answer is found, the message CORRECT is printed, and the program then branches from the subroutine to the next question presented in the main program. If the answer is not found, the message INCORRECT, PLEASE TRY AGAIN is printed, and the GOTO in Line 1040 sends the program back to Line 900 so the student can reenter another answer.

This program illustrates the efficiency and effectiveness of using a subroutine whenever possible. Subroutines usually result in shorter programs that are easier to code and follow.

---

1 Why is a string search used in an educational computer program?

2 Which string function should be used to determine the number of iterations necessary to search for a particular string in a student's input? Explain how this function determines the appropriate number of iterations.

3 Which string function should be used to search a student's input for a particular string? Explain how this function accomplishes the search.

4 Describe the relationship of the LEN function and the MID$ function in searching a student's input for a particular string.

5 Write a short program that requests the user to enter his or her name and then prints the first letter of the name on one line, the first and second letters on the second line, the first, second, and third letters on the third line, and so on, until the full name is printed on the last line. Use a FOR/NEXT loop together with the LEN and MID$ functions to accomplish the task.

Keyboard Control 4.8

You are probably familiar with the use of INPUT and GET to interrupt the run of a program so that information may be entered that will be used in the subsequent execution of the program. For example, a GET A$ statement may be used in conjunction with the prompt DO YOU WANT TO SEE THE MENU? (Y/N). If the user types a Y, a statement that follows, such as IF A$ = "Y" THEN 200, sends the program to a routine that displays the menu frame on the screen.

INPUT and GET statements do allow some keyboard control of the run of a program. But when these statements are executed, the program comes to a halt and does not continue until an appropriate keyboard entry is made. Thus, these statements do not provide real time control while the program is running. The simple keyboard control routine described in this section allows control of a program run based on pressing a single key designated within the program. The next program illustrates the keyboard control routine. It begins by displaying this question on the video screen: GUESS WHAT KEY YOU MUST PRESS TO MAKE ME PRINT A MESSAGE?. The program executes Lines 80–90 in a continuous loop produced by the THEN 80 statement in Line 90. To break out of the loop, the user must press the X key. When X is pressed, the screen is cleared and the message THIS LINE IS PRINTED WHEN YOU PRESS X appears. Pressing any key other than X has no effect; therefore, the run of the program is controlled by the user by the X keystroke.

```
30  HOME
40  VTAB 10
50  PRINT "GUESS WHAT KEY YOU MUST PRESS TO MAKE ME PRINT A MESSAGE?"
70  PRINT "CONTINUE TO PRESS A DIFFERENT KEY UNTIL YOU PRESS THE CORRECT
     KEY."
80 C =  PEEK (49152)
90  IF C <  > 216 THEN 80
100  POKE  - 16368,0
510  HOME
520  VTAB 10
527  PRINT "MESSAGE:"
532  PRINT "THIS LINE IS PRINTED WHEN YOU PRESS X."


]RUN
GUESS WHAT KEY YOU MUST PRESS TO MAKE ME PRINT A MESSAGE?
```

```
CONTINUE TO PRESS A DIFFERENT KEY UNTIL YOU PRESS THE CORRECT KEY.
MESSAGE:
THIS LINE IS PRINTED WHEN YOU PRESS X.
```

We now explain how the keyboard control routine works. The routine is made up of two lines (Lines 80 and 90) that form a continuous loop. In Line 80 the value stored in memory location 49152 is assigned to the numeric variable C. Memory location 49152 in the Apple II holds an ASCII code based number indicating the last key on the keyboard that was pressed. As the program goes through the loop formed by Lines 80 and 90, the value of the last key pressed is stored in memory location 49152. In Line 80 this value is assigned to C. The number is the ASCII value of the character for the key plus 128. If B is pressed, for example, the value stored at memory location 49152 is 194 (the ASCII value of B, which is 66, plus 128).

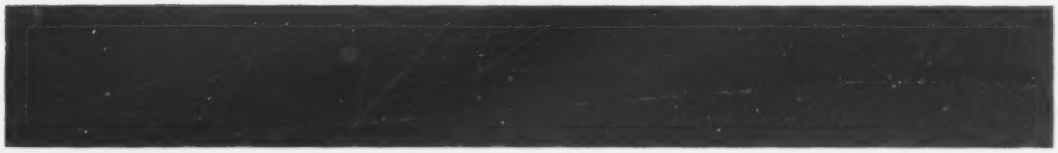


Figure 4.4
This subroutine prints (PRESS SPACE BAR TO CONTINUE) near the bottom of the screen. The loop formed by Lines 2210 and 2220 continually checks for a key press. If the Space Bar is pressed, the GOTO 2230 statement causes the program to exit the loop to the RETURN statement in Line 2230, and the run returns to the main program.

The value stored when the X key is pressed is 216 (88 + 128 = 216). If C < > 216 (Line 90), the program branches back to Line 80. The loop formed by Lines 80 and 90 is continuous until the X key is pressed. When the X key is pressed, the value stored at memory location 49152 is 216, and 216 is assigned to C in Line 80. Since the condition in Line 90 is false, control goes to Line 100. POKE -16368,0 in Line 100 clears the keyboard for another key press. Lines 527 and 532 print a message on the screen, indicating the X key has been pressed.

Finding ways to incorporate a keyboard control routine in an educational program should stimulate your imagination and creative juices. For example, the foregoing program can be adapted into a simple game format to be used in teaching the concept of ASCII code numbers. Add Lines 35, 65, 75, and 533, and modify Lines 90 and 532 as shown:

```
35 K =  INT (26 *  RND (1) + 65)
65  PRINT "CLUE: ";K
75 AK = K + 128
90  IF C (  ) AK THEN 80
532  PRINT "YOU PRESSED THE "; CHR$ (K);" KEY."
533  PRINT "THE ASCII VALUE OF THE LETTER IS:  ";K
```

When the modified program is run, the question GUESS WHAT KEY YOU MUST PRESS TO MAKE ME PRINT A MESSAGE is printed. Also, a clue, the ASCII number for the letter of the key the user should press to see the message, is printed. For example, if F is the key to press, the screen will display CLUE: 70, since 70 is the ASCII value for F. The ASCII code number selected by the program is based on the statement in Line 35. Line 35 generates a random number from 65 through 90, the ASCII code numbers for the uppercase letters of the alphabet, and assigns the number to the variable K. Line 65 prints this number (K) as a clue to guess the correct key to press. In Line 75, AK is assigned the value of K plus 128 to conform to the number that will be stored in memory location 49152.

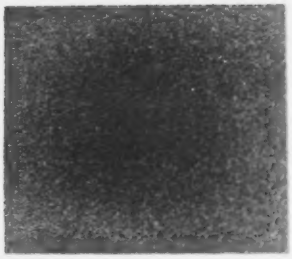
When the key with the letter whose ASCII value equals K is pressed, the IF condition in Line 90 is not true. The program breaks out of the loop formed by Lines 80 and 90 and control goes to Line 100. Line 532 prints a message to remind the user which key has been pressed. CHR$(K) is the code used to print the letter based on the ASCII number assigned to the variable K. Line 533 prints the ASCII number (K) of the letter.

Although the computer game described here is unlikely to rank among the "Top 40," it does illustrate the use of the random function coupled with a keyboard control routine. It should suggest ways in which you may be able to use the routine to enhance and enliven an educational computer program and perhaps even suggest an idea for a program you may wish to design and develop.

---

1 Write a routine that illustrates the use of INPUT to provide keyboard control for the user.

ON LINE 4.8



2 Write a routine that illustrates the use of GET for keyboard control.

3 How are ASCII code values related to keyboard characters?

4 Explain the keyboard control routine described on page 129.

5 Write a short game program that uses the keyboard control routine.

```
1290  REM  TELL USER THAT NUMBER MACHINE IS READY TO DISPLAY NUMBER
1300  VTAB 23: HTAB 10: PRINT "PRESS ANY KEY TO START";
1310  [illegible]
1320  [illegible]
1330  [illegible]
1340  [illegible]
1350  [illegible]
1360  [illegible]
1340  REM CLEAR KEYBOARD STROBE & MESSAGE ON BOTTOM OF SCREEN
1370  POKE - 16368,0: VTAB 23: CALL - 958
```

Figure 4.5
Lines 1310–1340 form a continuous loop that runs until the student presses a key. When a key is pressed, the condition in Line 1340 is met and the program breaks out of the loop. Line 1320 sounds the speaker while READY is printed on the screen, if the user has typed a Y at the beginning of the program to indicate that sound is desired (S$ = "Y").

4.9 Programming a Menu

A menu is a list of choices presented to the user of a computer program. The purpose of a menu in an educational computer program is twofold: to present a list of choices, and to transfer the user to the appropriate section or program. Just as a restaurant menu allows the customer to view the available dining selections, so too does

a computer program menu allow the user to view the program offerings. Given that most educational programs are intended for a broad spectrum of users, a menu driven program can offer flexibility both in pace and content. It can facilitate location of specific information and enable the user to omit viewing undesired sections. A menu may also allow the student to return immediately to a specific section at a later time or to locate desired information for review. A well-designed menu driven program enables all users to operate effectively an educational program for their individual needs.

Although menus vary in style, they all should describe the options available in nontechnical, brief phrases. A single keystroke, usually a number or letter, should access the desired selection. If your educational program is structured in such a way as to require submenus, then consistency in format is a must. For example, if you number your offerings 1, 2, and 3, then any submenu should also require numerical entries. In addition to listing the program topics, a menu should contain a quit option, which is a selection allowing the user to exit the program immediately. Needless to say, a menu requires accessibility from almost any frame in your program, thus providing yet another interactive mode so necessary in computer assisted instruction.

```
540  HOME : VTAB 4
550  PRINT  TAB( 14)"*** MENU ***"
560  PRINT : PRINT
570  PRINT  TAB( 14)"A. TUTORIAL"
580  PRINT : PRINT
590  PRINT  TAB( 14)"B. QUIZ"
600  PRINT : PRINT
610  PRINT  TAB( 14)"C. QUIT"
620  PRINT : PRINT : PRINT
630  PRINT  TAB( 12)"PRESS A, B, OR C."
640  GET K$
650  IF K$ = "A" THEN 690
660  IF K$ = "B" THEN 1690
670  IF K$ = "C" THEN 2170
680  IF K$ < > "A" OR K$ < > "B" OR K$ < > "C" THEN 540
```

Figure 4.6
Programming the menu for End Punctuation. Line 680 allows only A, B, or C as acceptable choices.

Programming a menu can be as simple as coding a number of PRINT statements, indented so that they are displayed in an orderly,

readable fashion. Figure 4.7 is an example of a simple menu that can be used in a drill and practice program:

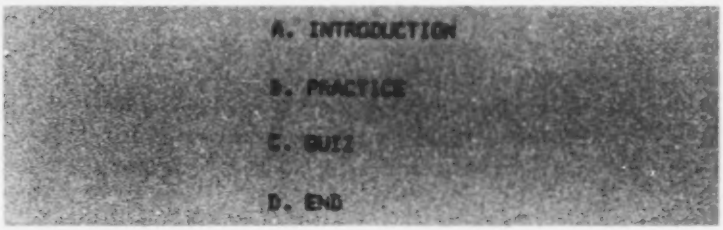


Figure 4.7
Example of a frame displaying a menu with four choices.

The programming of this frame is

```
10  HOME
20  PRINT : PRINT
30  PRINT  TAB( 8);"A. INTRODUCTION"
40  PRINT : PRINT
50  PRINT  TAB( 8);"B. PRACTICE"
60  PRINT : PRINT
70  PRINT  TAB( 8);"C. QUIZ"
80  PRINT : PRINT
90  PRINT  TAB( 8);"D. END"
```

Although this program segment displays a list of choices on the screen, it does not as yet provide for user selection of an item from the menu. To accept a selection by entering a single keystroke, programmers generally rely on the GET statement. Appending the previous code to include the following statements will enable the user to simply enter a letter and view the desired choice.

```
100  PRINT : PRINT : PRINT
110  PRINT  TAB( 7);"ENTER A, B, C OR D: ";
120  GET C$
130  IF C$ = "A" THEN 1000
140  IF C$ = "B" THEN 2000
150  IF C$ = "C" THEN 3000
160  IF C$ = "D" THEN  HOME : END
170  GOTO 120
1000  REM
2000  REM
3000  REM
```

Besides allowing for only a single keystroke, this routine "locks" the keyboard by rejecting any selection other than one of the four offered. The GOTO loop in Lines 120 to 170 insures that a listed

choice has been entered and maintains the screen display until the appropriate response is entered.

Except for the possible inclusion of a title or heading, the menu is complete. Inserting the program line below will complete the screen display first illustrated in Figure 4.7.

```
15  PRINT : PRINT  TAB( 7);"MINERAL IDENTIFICATION"
```

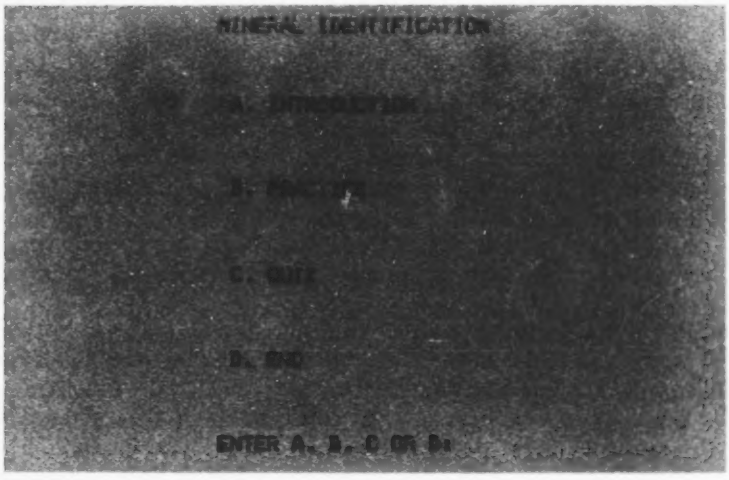


Figure 4.8
This menu identifies the program by name and directs the user to select and enter a choice of A through D.

When a valid selection is entered, program execution will branch to the appropriate portion of the program. Of course, the first line executed after the branch must contain a HOME. This will clear the menu from the screen and allow the selection to be presented.

It may be that your completed educational program is composed of several separate programs placed on the same disk. To access a separate program, you must load it from the disk into random access memory (RAM) before the run. To accomplish this, use the DOS (disk operating system) command, CHR$ (4), which represents Control-D. The following program lines assume that MINERAL QUIZ is a separate program and would be run if selection C in the previous menu routine is chosen.

```
150  IF C$ = "C" THEN 3000
3000  HOME
3010  PRINT : PRINT  CHR$ (4);"RUN MINERAL QUIZ"
```

The PRINT CHR$ (4) in Line 3010 is the DOS statement that allows the computer to literally read and execute the statement in quotations that immediately follows CHR$ (4). The first PRINT acts as a carriage return. DOS commands associated with specific titles cannot be executed unless the preceding action was followed by a carriage return. Thus, the extra PRINT insures this carriage return and is a safeguard that should be incorporated. Omitting a return before CHR$ (4) causes the statement to be printed in quotations, just as if the CHR$ (4) were not included.

This technique of running other programs on the disk by accessing them from a menu routine is generally referred to as "calling to the disk" and is used to expand the computer's memory by taking advantage of a disk's storage capability. Another advantage of running programs in one educational unit is the ease with which modular programming can be accomplished. If your program is to consist of four or five or even more units, it is much simpler to focus on one module of the entire program, perfect it, save it, and start coding a subsequent module. This technique guides the programmer into using a structured approach and lessens the unwarranted use of GOTO statements.

Once you have familiarized yourself with the fundamental use of a menu in a program and practiced the techniques used in coding such a menu, your creativity may very well initiate a desire for more ornate and elaborate menus. Depending on your style and programming knowledge, you may want to incorporate INVERSE, FLASH, sound, motion, or any of a variety of other stimulants into your menu program.

A preferred menu programming technique is outlined here. It incorporates the FLASH statement to inform the user of the selection made. Since the GET statement provides no output, a delay loop is used to verify to the user that the intended selection was the actual selection. This is done by flashing the letter of the menu selection for several seconds before the selection is actually executed.

```
10  HOME
20  FOR I = 1 TO 5
30  READ S$(I)
40  VTAB 2 * I: HTAB 10
50  PRINT S$(I)
60  NEXT I
70  DATA "  THIS IS THE TITLE","1. SELECTION #1","2.SELECTION #2","3. S
    ELECTION #3","4. END THE PROGRAM"
80  VTAB 13: HTAB 11
90  PRINT "ENTER 1, 2, 3 OR 4 ";
100  GET K$
```

```
110 K =  VAL (K$)
120  IF K < 1 OR K > 4 THEN 100
130  HTAB 10: VTAB 2 * K + 2
140  FLASH
150  PRINT K;"."
160  NORMAL
170  FOR P = 1 TO 3000: NEXT P
180  HOME
190  ON K GOSUB 1000,2000,3000
200  END
1000  REM  BRANCH TO SELECTION #1
2000  REM  BRANCH TO SELECTION #2
3000  REM  BRANCH TO SELECTION #3
```

Lines 20 through 70 read, format, and print the title and list of choices. The vertical tab in Line 40 double spaces the output. Line 80 centers the selection directions, and Line 90 prints them. Line 100 accepts the entry, and Line 110 converts it to its numerical value. If out of range, Line 120 rejects the entry, sending execution back to the GET statement in Line 100. Once a valid selection has been made, the selection number flashes on the screen for approximately 4 seconds. Line 130 sets both the horizontal and vertical TABs to the screen location preceding the intended selection. Line 140 sets the FLASH, and Line 150 prints the selection number over the identical existing number. Line 160 sets output mode back to NORMAL. The 4 second delay is provided by Line 170. After clearing the screen, the program branches to the desired choice. If the fourth selection, END, is made, then Line 200 is executed, because the ON GOSUB statement in Line 190 provides for only three choices, one less than the menu allows. Thus, selection number 4 would cause the program to end at Line 200.

As your educational computer programs increase in content, you will find a need to develop and incorporate menus. When this time arrives, your style will dictate the format of the menu routine. A simple, straightforward menu that lists the choices in clear, precise terms should be the top priority. This feature, coupled with the accessibility of the menu from almost any frame in your program, will increase both the educational and aesthetic value of the program.

---

1 What is the purpose of a menu in an educational program?

2 What choice is usually the last choice listed in a menu?

ON LINE 4.9

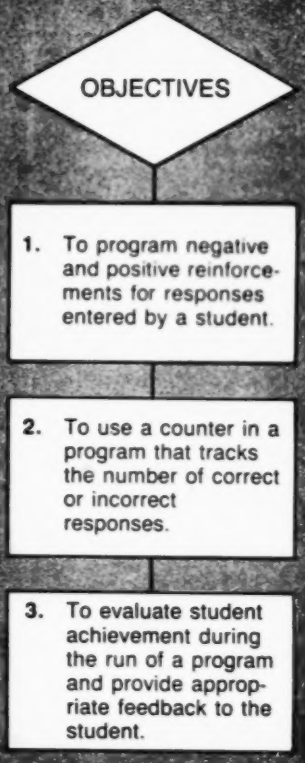
3 List three advantages of using a menu in an educational computer program.

4 Write a menu frame that allows the user to view a simple example, a complex example, an original example, or end.

5 Write a menu routine that inverses the name of the selection made when the user enters the corresponding number or letter of the selection.

CHAPTER FIVE

STUDENT ACHIEVEMENT


OBJECTIVES
1. To program negative and positive reinforcements for responses entered by a student.
2. To use a counter in a program that tracks the number of correct or incorrect responses.
3. To evaluate student achievement during the run of a program and provide appropriate feedback to the student.

During the run of an educational program, it is important for the student to find out how well he or she has learned the material. Therefore, the computer must count and tally the inputs entered by the student in response to questions and other prompts. Further, information as to whether or not a student response is correct must be given, usually in the form of positive and negative reinforcements—for example, the printing of CORRECT for the right answer and INCORRECT for the wrong answer. The tallied responses of the student must be evaluated in a meaningful way, and the evaluation printed on the computer screen as feedback to the student.

In this chapter you will learn how to code reinforcements and counters within the main part of a program and as subroutines. You will also learn how to count and tally correct and incorrect student responses within a program and how to program the evaluation of student achievement based on the student's responses.

5.1 Reinforcements

Reinforcements refer to feedback from an educational computer program that indicate to a student whether his or her responses to questions and prompts are correct or incorrect. Reinforcements can be divided into two groups: positive reinforcements and negative reinforcements. A positive reinforcement signals a correct response, and a negative reinforcement signals an incorrect response.

Consider a frame in a geography program that prints the question WHICH DIRECTION IS THE OPPOSITE OF NORTHWEST? If the student's input is the correct answer, SOUTHEAST, the computer should respond by printing a positive reinforcement such as CORRECT! or YOU GOT IT RIGHT. If the student enters an incorrect answer, the program should respond with a negative reinforcement such as SORRY, WRONG ANSWER. Reinforcements can also take the form of sounds (beep, buzz, or musical tone) or pictures (happy face, star, or merit badge).

To add interest to your program, you will want to include a variety of reinforcements. Generally, it is a good idea to use several different positive reinforcements while employing only one type of negative reinforcement. Why? Positive reinforcements are a reward for responding correctly. Therefore, the novelty and interest resulting from a variety of reinforcements should be associated with getting the answer right, not with getting it wrong. Feedback for an

incorrect input should be a simple acknowledgement such as INCORRECT.

Hangman is an example of a computer program that violates the foregoing principle of educational psychology. Each incorrect response using Hangman results in another addition to the building of the gallows, thereby increasing the interest value of the picture displayed on the screen. In other words, the screen display becomes more developed and interesting in direct relation to the number of incorrect entries.

The following program lines illustrate the use of reinforcements.

```
200  PRINT "HOW OLD WOULD FORMER PRESIDENT"
205  PRINT "JOHN KENNEDY BE IF HE LIVED TO THE YEAR 2000?
210  INPUT AN$
220  IF AN$ = "83" THEN  PRINT "CORRECT!": GOTO 240
230  PRINT "SORRY, WRONG ANSWER."
240  PRINT "**** PRESS SPACE BAR ****"
250  GET K$
```

```
]RUN
HOW OLD WOULD FORMER PRESIDENT
JOHN KENNEDY BE IF HE LIVED TO THE YEAR 2000?
?83
CORRECT!
**** PRESS SPACE BAR ****
```

Lines 200 and 205 print a question. The input answer (AN$) to the question is checked in Line 220, and the positive reinforcement CORRECT! is printed if the student gets it right. The GOTO that follows sends the program past Line 230 to Line 240, where the student is directed to press the Space Bar to continue the run of the program. If the student types the wrong answer, the condition in Line 220 is not satisfied, and the program goes to the following line (Line 230), which prints the negative reinforcement SORRY, WRONG ANSWER.

For adults and high school students the same reinforcements, CORRECT! and SORRY, WRONG ANSWER, may be used throughout a program. However, for younger students you may wish to use a variety of reinforcements. This can be done by reading a selected set of reinforcements into an array assigned to a subscripted variable and using a subroutine to print them. Study the following lines that show the programming involved in printing reinforcements with a subroutine.

```
50  FOR I = 1 TO 5
60  READ PR$(I)
70  NEXT I
75  DATA  CORRECT, RIGHT ANSWER, GOOD GOING!, O.K.- YOU GOT IT RIGHT, R
    IGHT!
100  PRINT "HOW MANY HOURS PAST NOON IS 1800 CLOCK  HOURS?"
110  INPUT AN$
120  IF AN$ = "6" THEN  GOSUB 5000
130  IF AN$ <  > "6" THEN  PRINT "SORRY, WRONG ANSWER."
4999  END
5000  REM  SUBROUTINE FOR PRINTING POSITIVE REINFORCEMENTS
5010 X =  INT ( RND (1) * 5) + 1
5020  PRINT PR$(X)
5030  RETURN

]RUN
HOW MANY HOURS PAST NOON IS 1800 CLOCK  HOURS?
?6
RIGHT!
```

Lines 50-70 constitute a FOR/NEXT loop that reads the data in Line 75 into the subscripted variable PR$(I). The data consists of five positive reinforcements. Line 100 prints a question, and Lines 120 and 130 check the answer entered in Line 110. If the correct answer, 6, is entered, Line 120 calls the subroutine beginning at Line 5000 and prints a randomly selected positive reinforcement. An incorrect answer causes the program to drop through Line 120 to Line 130, printing SORRY, WRONG ANSWER.

When the subroutine is called, it prints one of the five positive reinforcements stored in PR$(X). Line 5010 assigns a random number within the range of 1 through 5 to the variable X. In Line 5020 the positive reinforcement is printed based upon the value of X. For example, suppose 3 is generated as the random number. The positive reinforcement printed will be GOOD GOING! because it is the string value of PR$(3). What reinforcement would be printed if 5 is generated as the random number?

You may want to use your imagination in composing positive reinforcements and use more than five in your program. If so, you must change the DATA statement so that it contains the added positive reinforcements. Also, change the other lines needed to read the data into the subscripted variable PR$(I) and print out the reinforcements with the subroutine. For instance, if 10 positive reinforcements are in the DATA statement, the FOR statement in Line 50 must count to 10 (FOR I = 1 TO 10), and Line 5010 should be changed so that X will be assigned randomly a value from 1 through 10 (X = INT (RND(1) * 10 + 1)).

Some programs include positive reinforcements that reflect student jargon in vogue at the time the program is written. However, it is important to use good taste in selecting positive reinforcements. Avoid positive reinforcements that students might perceive as silly, corny, or condescending. It is equally important to select negative reinforcements that are not demeaning or sarcastic. One unsettled issue in educational computing is the use of sound as a reinforcement. Here you must use your best judgment in terms of the level of students for whom you are writing the program and the context in which the program is used. Most school teachers and librarians look with disfavor on any unwarranted and unnecessary noise being brought into the environment. Further, sound as reinforcement, particularly as negative reinforcement, indicates to others in the vicinity of the student user how well he or she is doing. For example, negative reinforcement sounds from a computer, such as a low pitched buzz when the question is missed, announce to classmates in the vicinity that the student is making mistakes. The situation may be embarrassing to the student and result in the student aborting the program. A reaction of this kind and other related problems will be avoided if you remember to use good taste and good sense when selecting reinforcements.

---

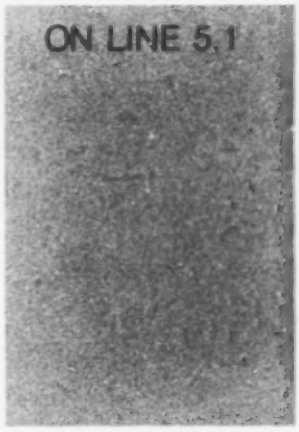
1 What are reinforcements?

2 What factors must be considered in selecting reinforcements for an educational computer program?

3 Why might several different positive reinforcements be used in a program for elementary school students?

4 Make a list of five positive reinforcements you would use in an educational computer program.

5 Write a program that asks a question, scores the answer entered by a student as correct or incorrect, and uses subroutines to print randomly selected positive and negative reinforcements.



Counters and Tallies 5.2

The computer is especially adept at counting and recording a tally of events in an educational program. Of particular concern in most

programs is the need to count the number of correct and incorrect responses to questions. Numeric variables are used in programs as counters. For example, if the variable containing the number of correct responses in the program is R, then each time a student gives a correct response to a question, R is increased (incremented) by 1. A simple statement in BASIC, placed in the appropriate line of the program, will accomplish the incrementation: R = R + 1.

At this point it is well to state what might be called a proverb of educational computer programming: a variable that is intended to count only the number of correct responses must not also inadvertently count incorrect responses. Easy, you say. Be advised that it is just such problems that give beginning and even some experienced programmers severe headaches. That is, the variable that is supposed to count correct responses is placed in an incorrect line of the program where it sometimes counts an incorrect response as correct.

Here is a simple program written to demonstrate the use of a variable in an educational program to count the number of correct student responses.

```
10   PRINT "PLEASE TYPE THE ANSWER FOR"
20   PRINT "EACH MATH PROBLEM."
30   PRINT
31  R = 0
40   PRINT "9 + 9 = ";
50   INPUT A
60   IF A = 18 THEN 90
70   PRINT "THE CORRECT ANSWER IS 18"
75   PRINT
80   GOTO 110
90   PRINT "VERY GOOD"
95  R = R + 1
100   PRINT
110   PRINT "12 + 4 = ";
120   INPUT A
130   IF A = 16 THEN 160
140   PRINT "SORRY, THE ANSWER IS 16"
145   PRINT
150   GOTO 180
160   PRINT "NICE WORK"
165  R = R + 1
170   PRINT
180   PRINT "7 + 5 = ";
190   INPUT A
200   IF A = 12 THEN 230
210   PRINT "NICE TRY, BUT THE ANSWER IS 12"
215   PRINT
220   GOTO 250
```

```
230  PRINT "YOU GOT IT"
235 R = R + 1
240  PRINT
250  PRINT "YOU HAD ";R;" CORRECT ANSWERS"
255  PRINT "OUT OF THREE PROBLEMS."
260  PRINT "IT WAS FUN WORKING WITH YOU."
265  END
```

The variable R is used to count and tally the number of correct answers and is initialized to zero in Line 31. R remains at zero until it is increased elsewhere in the program by the statement R = R + 1. Each time the student answers a question correctly, and only then, R is increased by 1. This means that when a student's answer is incorrect, R is not increased from its present value. The value of R continues to increase as the student answers questions correctly. Although our example program contains only three questions, the number of questions is actually irrelevant to the operation of the counter. You may include as many questions as you like. Just make certain that you also place a counter in an appropriate line so that each correct answer will be tallied.

Let's examine the program more closely. A problem, 9 + 9, is presented at Line 40, and Line 50 requests the student's answer. Line 60 checks the answer and, if correct, branches the program to Line 90, which prints VERY GOOD. Line 95 increments R to 1 (R = 0 + 1 = 1). Line 110 prints another problem, 12 + 4. If the answer entered at Line 50 is incorrect, the program branches to Line 70, which prints the correct answer. Line 75 gives an empty line, and then Line 80 branches the program to Line 110, thus skipping Line 95.

The sequence of events described here occurs after every problem is presented. A correct answer causes a branch to a line containing the counter, where the present value of R is increased by 1. An incorrect answer causes the program to bypass (skip) the line containing the counter. This is the crux of it! Add 1 to R when the answer is correct, and skip the line containing R when the answer is incorrect. With this procedure it is evident that R will contain the number of correct answers entered by the student on completing the program.

---

1 Why is it important to initialize a counter?

2 Why is it important for an educational computer program to keep track of the number of correct and incorrect responses entered by the user?

ON LINE 5.2

3 Explain how the number of wrong answers entered in an educational computer program can be determined by using the count of correct answers.

4 Write a short educational computer program containing five true and false statements to be answered by the user typing T or F. Include a counter that counts the number of correct responses and prints the total score at the end.

5.3 Counters and Tallies in Subroutines

If a counter is used often in a program, it may be more efficient to place it in a subroutine. Here is an example of how you would construct a subroutine to count both correct and incorrect answers.

```
20 R = 0:W = 0
30  PRINT "PLEASE ENTER TWO POSITIVE INTEGERS AND THEIR SUM."
40  FOR I = 1 TO 5
60  INPUT "ENTER THE FIRST NUMBER ";N1
62  INPUT "ENTER THE SECOND NUMBER ";N2
64  INPUT "ENTER THE SUM ";S
70 A = N1 + N2
80  GOSUB 1000
90  NEXT I
100  PRINT "DO YOU WISH TO SOLVE MORE PROBLEMS?(Y/N)"
120  INPUT A$
130  IF  LEFT$ (A$,1) = "Y" THEN 30
135 T = R + W
140  PRINT "YOU TRIED ";T;" ADDITION PROBLEMS."
150  PRINT "YOU HAD ";R;" CORRECT AND ";W;" INCORRECT ANSWERS."
160  PRINT "SEE YOU LATER."
165  END
990  REM  COUNTS NUMBER OF CORRECT AND INCORRECT ANSWERS
995  REM  AND PROVIDES APPROPRIATE FEEDBACK
1000  IF A = S THEN 1040
1010 W = W + 1
1020  PRINT "SORRY, THE ANSWER IS ";A
1030  GOTO 1060
1040 R = R + 1
1050  PRINT "CORRECT"
1060  RETURN
```

In Line 20, R and W are initialized to zero. Line 40 begins a loop that will be repeated five times. At Lines 60, 62, and 64 the student

enters two integers and their sum. For purposes of illustration, assume the student enters 2, 4, and 6. Line 70 computes the correct sum as 2 + 4 and stores 6 in the variable A. Line 80 then branches the program to the subroutine beginning at Line 1000, which determines if the student's answer is correct or incorrect, records the result, and prints an appropriate message. Line 1000 determines if the sum entered by the student is correct. In our example, 2 + 4 = 6 (A = S), the program branches to Line 1040, where R is incremented by 1. Then the message CORRECT is printed at Line 1050. Line 1060 branches the program to Line 90, the line immediately following Line 80, which called the subroutine. Line 90 branches to Line 40 for the second repeat of the FOR/NEXT loop.

If the answer entered by the student at Line 40 is incorrect, this answer would be noted as wrong at Line 1000 and the program would branch to Line 1010, where W is incremented by 1. Then at Line 1020 the message SORRY, THE ANSWER IS 6 is printed. Line 1030 branches the program to Line 1060, which branches the program to Line 90. Line 90 sends the program to Line 40 for another iteration of the loop.

After five iterations of the loop, five problems have been presented, and the program falls through to Line 100, which asks the student if more problems are desired. If the input is Y, the program branches to Line 40 and five more problems are presented. If the input is N, the program falls through to Line 135, which computes the total number of problems attempted by summing the number of correct and incorrect answers. Line 140 informs the student of the number of problems attempted, and Line 150 gives the number of correct and incorrect answers.

Note that the subroutine listed in this program can be used at any time from within the main program. You may include as many different FOR/NEXT loops as you wish, with each loop giving different types of problems and requesting student input. The subroutine can be called from all of these loops and used over and over again. Think of the savings in terms of program lines. Instead of writing the routine for finding correct and incorrect answers for each of the many different loops, you need only write it once and place it in a subroutine.

Write a program that contains three multiple choice questions. Include a subroutine to count and print the number of the user's correct responses.

5.4 Counting Entries

Counters can tally more than just the total of a student's correct and incorrect answers to questions. For example, perhaps you may wish to give the student two attempts to come up with the correct response to a question. Study the next program to see how this is done. We use the variable R to count correct answers and the variable A1 to count the number of student attempts for each question.

```
10  PRINT "NAME THE CAPITALS OF THE FOUR STATES "
15  PRINT "WHICH BEGIN WITH THE SAME LETTER AS"
18  PRINT "THE STATE."
20  PRINT "YOU WILL BE GIVEN TWO ATTEMPTS TO NAME"
25  PRINT "EACH CAPITAL. THE COMPUTER WILL PRINT"
28  PRINT "YOUR SCORE."
60 R = 0
70  FOR I = 1 TO 4
75 A1 = 0
80  READ S$,C$
90  PRINT S$;" ";
100  INPUT A$
110  IF A$ = C$ THEN R = R + 1: PRINT "CORRECT": GOTO 160
120 A1 = A1 + 1
130  IF A1 < 2 THEN  PRINT "PLEASE TRY AGAIN.": GOTO 90
140  PRINT "THE CAPITAL OF "S$;" IS ";C$;"."
160  NEXT I
170  PRINT "YOUR SCORE IS "; INT (R / 4 * 100 + 0.5);"%"
180  PRINT "HAVE A NICE DAY."
190  DATA "DELAWARE","DOVER"
200  DATA "INDIANA","INDIANAPOLIS"
210  DATA "HAWAII","HONOLULU"
220  DATA "OKLAHOMA","OKLAHOMA CITY"
340  END
```

R is initialized to zero outside the FOR/NEXT loop that begins at Line 70 and that will repeat four times, one repetition for each state. Line 80 reads the name of a state into S$ and the name of a capital into C$. Line 90 prints the name of a state followed by a blank so that the name of the capital entered by the student at Line 100 will be appropriately spaced beside the name of the state. Line 110 contains three statements separated by colons. If the student enters the correct capital at Line 100, A$ will equal C$, and R will be incremented by 1 in Line 110. At this point some programmers mistakenly believe that the program will then fall through to Line 120 and not execute the remaining statements in Line 110. Not true! When the IF-THEN condition in Line 110 is true, all remaining statements in Line 110 will be executed. The message CORRECT will

be printed, and then the program will branch to Line 160, thereby bypassing Lines 120–140 and moving into a second iteration of the loop.

Let's return to Line 100 and assume this time that the student initially enters an incorrect answer. The program would fall through Line 110 and execute 120, incrementing A1 by 1, which indicates an incorrect response on the first attempt. At Line 130, since A1 is less than 2 (1 < 2), the message PLEASE TRY AGAIN is printed and the program branches to Line 90, where the name of the same state is again printed. The student now enters a second attempt to name the capital of this state at Line 100. If the second attempt is correct, A$ = C$ in Line 110, R is increased by 1, the message CORRECT is printed, and the program branches to Line 160. If the second attempt is incorrect, A1 will be increased to 2, the IF-THEN condition at Line 130 will be false, the program will drop through to Line 140, print the correct answer, and then branch to Line 70 for another iteration of the loop. Note carefully that A1 is always reinitialized to zero at Line 75 for each iteration of the loop.

Study Line 170, which gives the student's score after having completed the program. Since R is the number of correct answers and 4 is the total number of states presented, R divided by 4 multiplied by 100 is the percent correct. We added 0.5 to this so that the computer would print this result to the nearest whole percent regardless of the number of states presented in your program. For example, if the program presented 15 states and the student gave 7 correct answers the computed percent would be approximately 46.666 percent and INT of this value is 46. By adding the 0.5 to 46.666 first, the result would be approximately 47.166, and INT of this value is 47, the integral percent.

This section presented several methods of counting and recording a tally of events, such as correct responses and number of attempts provided for answering questions. Now is the time for you to exercise your creative bent and invent some of your own programs that include a mixture of these methods designed to make your program more efficient and effective.

ON LINE 5.4

1 Write a program containing one multiple choice question in which the user must select the correct answer from four choices. The program should allow the user four opportunities to answer correctly and provide appropriate feedback when an incorrect answer is selected.

2 Add a counter to the program for question 1 that keeps track of the number of responses the user makes in answering the question correctly.

3 Write a program that contains three multiple choice questions. Include a subroutine that allows the user two chances to answer each question correctly. If the correct answer is not selected after the second response, then inform the user of the correct answer.

4 Add a routine to the program for question 3 that prints a summary as follows:

```
NUMBER OF QUESTIONS CORRECT ON FIRST TRY:
NUMBER OF QUESTIONS CORRECT ON SECOND TRY:
NUMBER OF QUESTIONS INCORRECT AFTER TWO
ATTEMPTS:
```

5.5 Evaluation of Student Achievement

Although counters and tallies, as discussed in Sections 5.2 and 5.3, are necessary and important, they do not represent an evaluation of the student's achievement. Suppose the performance objective for a program is that the student will be able to add two positive integers. How many such additions must the student perform correctly to indicate achievement of the objective? An evaluation such as this requires more than a tally of the number of correct and incorrect responses given by the student. Evaluation requires a subjective element; that is, some judgment must also be made in terms of how many correct answers are required to indicate inadequate, minimal, average, above average, or superior performance. The computer cannot make such a value judgment. Only you, the programmer, can. After you have done so, the computer can implement the judgment in providing appropriate feedback to the student.

You must decide what constitutes various levels of student performance and determine the appropriate type or form of evaluative feedback that the computer is to give the student, based on his or her performance. Evaluation should, in part, depend on the age level and maturation level of the students using the program. For more mature students the computer might print a simple message such as VERY GOOD, NICE WORK, or KEEP TRYING. For younger

students a "happy" face could be displayed on the screen to indicate stronger performance, and a "sad" face could indicate poorer performance. Using both buzzes and beeps together with pertinent graphic displays on the screen would serve to provide even stronger evaluation of student performance.

Here is an enhanced version of a program presented on page 146 that requests the student to enter and add two positive integers. That is, the student is encouraged to make up problems and then solve them.

```
30  PRINT "PLEASE ENTER TWO POSITIVE INTEGERS AND THEIR SUM."
40  FOR I = 1 TO 5
60  INPUT "ENTER THE FIRST NUMBER ";N1
62  INPUT "ENTER THE SECOND NUMBER ";N2
64  INPUT "ENTER THE SUM ";S
70 SU = N1 + N2
80  GOSUB 1000
90  NEXT I
100  PRINT "DO YOU WISH TO SOLVE MORE PROBLEMS(Y/N)?"
120  INPUT A$
130  IF  LEFT$ (A$,1) = "Y" THEN 30
135 T = R + W
140  PRINT "YOU TRIED ";T;" ADDITION PROBLEMS."
150  PRINT "YOU HAD ";R;" CORRECT AND ";W;" INCORRECT ANSWERS."
160 SC =  INT (R / T * 100 + .5)
170  PRINT "YOUR SCORE IS ";SC;"%"
250  PRINT "SEE YOU LATER."
300  END
990  REM  COUNTS NUMBER OF CORRECT AND INCORRECT ANSWERS
995  REM  AND PROVIDES APPROPRIATE FEEDBACK
1000  IF S = SU THEN 1040
1010 W = W + 1
1020  PRINT "SORRY, THE ANSWER IS ";SU
1030  GOTO 1060
1040 R = R + 1
1050  PRINT "CORRECT"
1060  RETURN
```

Note that the computer gives the student an opportunity to work as many sets of five problems (Lines 40 and 100) as desired. At the end of the program the student is informed of the number of addition problems attempted, the number of correct answers, the number of incorrect answers, and the percent correct. But no evaluation is provided concerning whether the student's achievement of the objective was inadequate, minimal, average, or superior. Only after a value judgment is made by the programmer as to what these words

mean in terms of the student's score can any evaluation be provided to the student by the computer.

Assume the programmer decides as follows: a score of 90 or above means excellent work; a score between 80 and 90 is very good; a score between 70 and 80 is average; a score between 60 and 70 suggests that the student should possibly work through some more addition problems; and a score of less than 60 means the student should seek additional instruction on the addition of two positive integers elsewhere and then return to the computer program to try some more addition problems. With this judgment in hand it is not difficult to add some statements to our program so that the student will receive an appropriate evaluation.

```
180  IF SC >  = 90 THEN PRINT "EXCELLENT": GOTO 250
190  IF SC >  = 80 THEN PRINT "VERY GOOD": GOTO 250
200  IF SC >  = 70 THEN PRINT "SATISFACTORY": GOTO 250
210  IF SC >  = 60 THEN PRINT "YOU MAY WISH TO RUN THE PROGRAM AGAIN": GOTO
     250
220  PRINT "PLEASE SEE YOUR TEACHER FOR MORE HELP IN ADDITION."
```

The added lines check to determine in what range the student's score falls, and they provide appropriate feedback based on this range.

Figure 5.1
Programming of the evaluation feedback for End Punctuation based on a score of 1 to 10 questions answered correctly.

Sound is another evaluative technique that can be added to an educational program through use of the Apple's speaker. You can control the speaker with BASIC statements that make the speaker produce clicks, beeps, buzzes, and other interesting sounds and tones. One kind of sound is based on memory location—16336. It is connected to the speaker in such a way that when a BASIC program reads it with a PEEK function, the speaker makes a brief, almost inaudible, click. You can generate different sounds and tones by repeatedly reading this location, something like plucking the strings on a guitar or banjo: the faster the speaker is plucked using the PEEK function, the higher the frequency of the tone produced.

Type and run this one line program.

```
100 P =  PEEK ( - 16336)
```

If you listen closely you will hear a very faint, solitary click. We certainly can't do much with this! To produce some kind of continuous sound, we must pluck the speaker repeatedly. Let's revise Line 100 as follows:

```
100 P =  PEEK ( - 16336): GOTO 100
```

Run this program until it tires your ears. Then push Control-C to stop the program. Actually the sound produced is not very interesting since it is a continuous buzz. The program simply plucks the speaker as fast as is possible with BASIC statements. The range of tones you can produce with Applesoft BASIC is rather limited.

It is easy to control the duration of whatever sound the speaker is emitting. Use FOR/NEXT statements to accomplish this. Type and run this short program.

```
100  FOR I = 1 TO 250:P =  PEEK ( - 16336): NEXT I
```

Well, the sound is about the same. But at least we are able to control the length of time the speaker is buzzing. Simply change the present value of 250 in the loop to a smaller or larger value to decrease or increase the duration of the sound. We can use this buzz as an audio indicator of a student's incorrect answers in an educational program. And we can use the CHR$ function to produce a nice sounding beep to indicate correct answers. The CHR$ function returns the ASCII character that corresponds to the value in the argument. If a 7 is placed in the argument, a beep rather than a character is returned. Type and run this one line program.

```
110  PRINT  CHR$ (7)
```

Did you hear the beep? The 7 is the ASCII code for the bell, a mechanical contrivance on outdated Teletype terminals used years ago. Since the Apple does not have a bell, the speaker emits a beep sound instead.

Let's revise Line 110 to create a program that will give five successive beeps.

```
110  FOR I = 1 TO 5: PRINT  CHR$ (7): NEXT I
```

Not bad! We can use this five beat sound to provide feedback for correct responses and the buzz described earlier to provide feedback for incorrect responses. Let's return to our program on addition of positive integers and revise it to provide the required audio feedback. Add the following lines to the subroutine:

```
1011  FOR I = 1 TO 250:P =  PEEK ( - 16336): NEXT I
1041  FOR I = 1 TO 5: PRINT  CHR$ (7): NEXT I
```

Now rerun the program. The computer will sound the beeper five times when an answer is correct and then print the message CORRECT. The computer will sound the buzzer for a short time when an answer is incorrect and then print the message SORRY, THE ANSWER IS followed by the correct answer. If desired, one could also include some character graphic display in the same program to give additional feedback.

You must be the judge of not only the type of feedback to include in your educational programs but also how much feedback is appropriate. There are some educators who feel that little negative feedback should be included in educational programs. Ingenuity can be applied in tailoring positive reinforcements and motivational features to the purposes of a particular program without including negative reinforcements. One teacher with whom the authors have worked has written a drill and practice program entitled Slaying the Arithmetic Dragon, which is designed to help students review basic math skills. A graphic display of a dragon is presented early in the program, and students do sets of 10 math problems. Successful completion of the problems results in the eventual demise of the dragon, rather vividly portrayed in another graphic. Because students who use this program have such poor math skills, negative feedback (reinforcement) is deliberately omitted. The aim is to include nothing that might lower the student's self-esteem. A positive reinforcement is displayed on the screen after a problem is answered correctly. If a problem is answered incorrectly the first time but correctly the second time, it does not count as an error. When a student completes 9 out of 10 problems correctly in a set, he or she is rewarded by being allowed to play a stimulating game that is part of the Dragon program.

Including negative reinforcements in your programs is a matter only you can decide. We suggest that you carefully consider the intended audience of your programs. If the audience is immature, of very low ability, and would be easily hindered in their progress or offended by certain types of feedback, no matter how well intentioned, then such reinforcements should be omitted. Another option is to program user control of sound, allowing the student to turn it off at the beginning of a program.

ON LINE 5.5

1 What kind of feedback should be given the user of an educational computer program? Explain your answer.

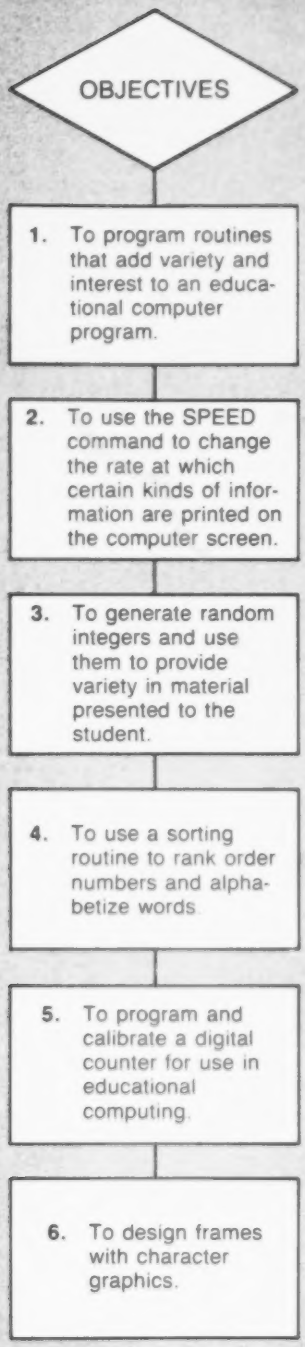
2 Should both positive and negative feedback be given to a user's responses in an educational computer program? Support your answer in terms of both a psychological and pedagogical position.

3 Why is evaluation both an objective and subjective process?

4 Write a short educational computer program that asks the user to identify the works of three famous authors. Include feedback and an evaluation of the user's responses.

CHAPTER SIX

VARIETY AND INTEREST


OBJECTIVES
1. To program routines that add variety and interest to an educational computer program.
2. To use the SPEED command to change the rate at which certain kinds of information are printed on the computer screen.
3. To generate random integers and use them to provide variety in material presented to the student.
4. To use a sorting routine to rank order numbers and alphabetize words.
5. To program and calibrate a digital counter for use in educational computing.
6. To design frames with character graphics.

Many routines may be developed that add variety and interest to a program. Foremost among these is the attention getting animation of text on the computer screen and changes in the speed at which text is printed. This chapter explains these features along with use of the random function for selecting questions and reinforcements randomly and creating other random features that add vitality to a program. Using a bubble sort to rank order numbers and alphabetize words is explained in Section 6.5. Creating a digital counter for use as a timer in programs that require this feature is also covered. The development of screens that contain character graphics, such as a happy face, is discussed in detail.

6.1 Text Motion

Motion of text on the video screen is a feature that can distinguish and enhance an educational computer program. Text motion is defined here as movement of printed material about the screen. It can be used to create visual appeal in a frame that introduces a program segment. For example, rather than simply printing ADDITION QUIZ on the screen to introduce that portion of a program, the programmer may opt to have this phrase printed in "ticker tape" fashion using the following code.

```
5 A$ = "              ADDITION QUIZ "
10  FOR I = 1 TO  LEN (A$)
15  VTAB 7: HTAB 15: PRINT  MID$ (A$,I,13)
20  FOR P = 1 TO 150: NEXT P
25  NEXT I
```

An examination of the following line by line description will provide insight into the creation of this illusion of text motion.

5 Creates the phrase that is to be moved across the screen. The 13 blank spaces preceding the phrase ADDITION QUIZ provide for gradual entry into the viewing range, and the blank space following the phrase allows for gradual exit.

10 Counter in the FOR/NEXT loop. The number of iterations, 27, is determined by the length of A$, which is 27 characters or blanks.

15 Positions the phrase at a specific starting location and prints 13 consecutive characters, including blank spaces, beginning at the designated location.

20 Provides a short delay so that the phrase will "move" at a readable speed.
25 Returns to Line 10 to increment the substring of the phrase to be printed.

The first time through the loop, the 13 leading blank spaces would be displayed. After a short delay (Line 20), 13 characters (or spaces), starting with the second position, will be viewed. This would consist of 12 blanks and the letter A. After another delay, the string "AD" will be printed in the same location set by the VTAB and HTAB in Line 15. This continues until the entire phrase ADDITION QUIZ appears on the screen. Eventually this phrase disappears, and once again only 13 blank spaces are displayed. The movement is created in Line 15, where the MID$ function partitions the phrase and is assigned a specific number of characters (13 in this example) starting at the Ith position. When I is incremented by 1 and the section of the phrase created by MID$(A$,I,13) is printed in the same location, the 13 characters to be displayed are shifted over one character.

The final 13 blank spaces that follow the ticker tape phrase are created by inserting a single trailing space in A$. This space prints over the last character, Z, as the routine goes through its last 13 iterations. Omitting the trailing space will eventually produce a row of 13 Zs as the MID$ function forms valid substrings consisting of all remaining characters, including blanks, when the desired length of the substring to be produced is greater than the number of characters remaining in the original string. Thus the absence of the trailing blank space would produce the following substrings as the loop concluded:

```
QUIZ
UIZ
IZ
Z
```

Since the printing of these substrings would originate in the same starting location, the final result would be ZZZZZZZZZZZZZ. The insertion of a trailing blank eliminates the Z and provides for the gradual disappearance of the entire printed phrase.

A frame with text movement can be enhanced by using the screen borders discussed in Section 3.7. Including the following lines in the preceding program provides a marquee effect to the text motion.

```
1  VTAB 6: HTAB 13: PRINT "******************"
2  VTAB 7: HTAB 13: PRINT "**                **"
3  VTAB 8: HTAB 13: PRINT "******************"
```

Before examining a more elaborate movement routine, experiment with this first program to develop your understanding of the programming involved. Try entering different phrases for A$, being sure to make any necessary changes for the last value in the MID$ function. Then try altering the location of the movement by changing the VTAB and HTAB values. Next, delete the space after the phrase in Line 5 and run the program. The results will coincide with those already discussed. Experimenting with the delay loop in Line 20 will also help provide the information you need to use this routine effectively in one of your programs.

Once the text motion programming technique is understood, the challenge of creating your own text motion routine for inclusion in a program can be as simple as changing the phrase provided in the foregoing example. For many programmers, however, creative desires impel them to expand on the uses of text motion. If you wish to delve further into the text motion aspect of programming, we recommend that you enter the following program into your computer and run it. You will observe the word ACCELERATION moving backward across the screen, apparently hitting the left side of the screen, dropping one row and moving forward. All of this occurs as the word gains speed, or accelerates, as it travels across the monitor.

```
80  HOME
85  VTAB 4: HTAB 15: PRINT "WELCOME TO"
90 A$ = "                                        NOITARELECCA "
95 B$ = "                    ACCELERATION"
100  FOR I = 1 TO 66
105  VTAB 5: HTAB 1: PRINT  MID$ (A$,I,40)
110  IF I > 41 THEN  VTAB 6: HTAB 1: PRINT  RIGHT$ (B$,I - 41)
115  FOR P = 1 TO 350 - 5 * I: NEXT P
120  NEXT I
```

A line by line description of this program shows that the programming previously illustrated with The Addition Quiz routine is once again at the heart of the text movement.

80 Clears the screen.
85 Sets the tabs and prints the first part of the title page.
90 Assigns the word to be moved (backward) to the variable A$. 40 spaces precede the word.

95 Assigns the word to be moved (forward) to the variable B$. 17 spaces precede the word.
100 Counter in the FOR/NEXT loop. Since the word will move across the entire screen (40 columns) and halfway back, the number of iterations should be approximately 60, depending on final placement of the word.
105 Positions the phrase at a specific starting location and prints the first 40 characters, or spaces, beginning at that location.
110 Checks to see if the word has traveled across the screen. If so, the phrase in B$ is positioned at a specific starting location and the rightmost characters of that phrase are printed. The number of characters printed is determined by the value of I − 41.
115 Provides a short delay that is decreased as the value of I is increased.
120 Returns to Line 100 to increment both the section of the phrase to be printed and the speed at which it is to be moved.

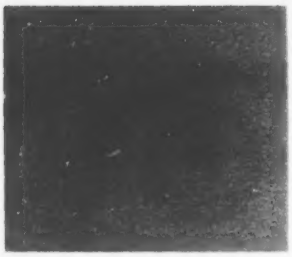
The 40 spaces preceding NOITARELECCA ("acceleration" spelled backward) and the number 40 in MID$(A$,I,40) allow the word to move across the entire 40 column text screen. Once the first letter of the word disappears from the screen (I > 41), the word stored in B$, ACCELERATION, is activated. The first printing of RIGHT$(B$,I − 41) involves only the rightmost character, the N that disappeared from the previous text line. The next iteration moves another character in A$ off the left side of the screen, and the last two letters of B$ are printed beneath that section of A$ still on the screen. The values in the expression 350 − 5*I in the delay loop (Line 115) are determined through experimentation. You might want to change this expression to suit the needs of your program.

Use of the text motion on the ACCELERATION title page adds more than variety and interest; the movement actually illustrates the concept being studied! Acceleration of the word itself prepares the student for the upcoming programmed lesson. With the foregoing examples in mind, you can use your imagination and creativity to generate various forms of text motion that will stimulate enthusiasm and learning among students.

---

1 Write a program that moves the letter *M* across the screen.

ON LINE 6.1



2 Write a program that moves your name across the bottom of the screen.

3 Change the program in question 2 so that your name appears in INVERSE mode as it moves across the screen.

4 Change the Acceleration routine on page 160 to move the word DECELERATION across the screen. Be sure that the word slows down as it moves across the screen.

6.2 Spinner

Sometimes in your educational computer program you may wish to direct the student's attention to a particular location on the video screen. One way to do this is by printing a 1 at this location and making the 1 appear to spin. The rapid rotation of the 1 as it spins around attracts the viewer's attention. Appropriately, we will call the spinning 1 a "spinner." You can print a spinner anywhere on the screen by inserting the following routine into your program.

```
5  HOME
10 S$ = "1/-"
20  FOR I = 1 TO 75
30  FOR J = 1 TO 3
40  VTAB 10: HTAB 20
50  PRINT  MID$ (S$,J,1)
60  VTAB 10: HTAB 20
65  PRINT " "
70  NEXT J
80  NEXT I
```

Line 10 assigns three characters, 1/−, as the value of the string variable S$. The FOR/NEXT counter in Line 20 causes the inner nested loop (Lines 30–70) to be executed 75 times. The inner loop, beginning with Line 30, is coded to count from 1 to 3, so each character in S$ will be printed in succession at the same point on the screen by Line 50 as the loop runs. Since J in Line 50 will have a value, successively, of 1, 2, and 3, as the inner loop iterates, the first character (1), second character (/), and third character (−) in S$ are printed successively at the point specified by Line 40: Row 10 and Column 20. Line 65 prints a space, blanking out each character immediately after it is printed. The spinner works by printing a 1 and blanking it out. Next a / is printed and blanked out. Then a − is printed and

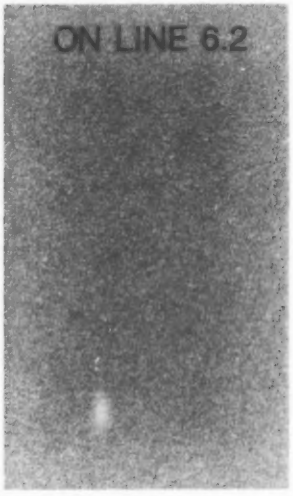
blanked out. The repeated printing and blanking out of these characters makes the 1 appear to rotate about a point because the characters / and – are, respectively, at 45° and 90° angles to the vertical 1. The apparent motion created is similar to the animation produced by the moving frames of a motion picture. Each run through the inner loop (FOR J = 1 TO 3) causes the 1 to "rotate" once. When the spinner routine is run, the 1 will "rotate" 75 times because the counter (FOR I = 1 to 75) in Line 20 causes the inner FOR/NEXT loop to repeat 75 times.

To emphasize material printed on the screen, place the spinner alongside a key word or term to which the student should pay extra attention. Use HTAB and VTAB to locate the point on the screen for printing. If you use the spinner more than once in a program, put it in a subroutine. The following listing shows the use of a subroutine to print the spinner three places to the left of the phrase HOUSE IS A NOUN. The values of V and H are set in the main program so that the spinner will be printed at the VTAB and HTAB locations specified by these values in subroutine Lines 5030 and 5045.

```
190  HOME
200 V = 5:H = 10
210  VTAB V: HTAB 13
215  PRINT "HOUSE IS A NOUN"
220  GOSUB 5000
225 V = 8
230  VTAB V: HTAB 13
240  PRINT "TREE IS A NOUN"
250  GOSUB 5000
4950  END
5000 S$ = "1/-"
5010  FOR I = 1 TO 75
5020  FOR J = 1 TO 3
5030  VTAB V: HTAB H
5040  PRINT  MID$ (S$,J,1)
5045  VTAB V: HTAB H
5050  PRINT " "
5060  NEXT J
5070  NEXT I
5080  RETURN
```

When the program is run, Line 215 prints the phrase HOUSE IS A NOUN at the location set in Line 210, Row 5 and Column 13. Line 220 calls the subroutine at Line 5000 that prints the spinner at VTAB 10, left of the phrase. Line 240 prints the phrase TREE IS A NOUN at Row 8 (V = 8), Column 13, as specified by the coordinates set in Line 230. Line 250 calls the subroutine again to print the spinner at the left of this phrase.

If your program is to be created and used on the Apple IIe or IIc, you will observe a smoother rotation of the spinner if you add to S$ the backslash character (\ ) that these machines are capable of printing. On the Apple IIe and IIc you can also use a pipe (|) instead of 1 (S$ = "|/-\ "). The counter for the FOR/NEXT loop should be FOR I = 1 TO 4 because there are now four characters in S$. The backslash and pipe are not available on the Apple II or Apple II+ keyboards.



1 Explain how the spinning effect is created in the program listed on page 162.

2 Write a program in which INTRODUCTION is printed in the center of the computer screen and a spinner is printed to the left of the word.

3 Enhance the program written for question 2 so that a spinner is also printed to the right of INTRODUCTION.

4 ASCII code number 92 can be used to print a \ on the Apple II+. Rewrite the spinner routine to include PRINT CHR$(92) so that a \ is printed after the - is printed in the same position as the other characters each time the inner loop is executed. Input the revised routine into your computer and run it. How does the run of the revised routine compare with the original routine?

6.3 Speed Change

The Apple computer has a command you can use to adjust the rate at which printed material is presented on the screen. This command, SPEED, can be set to any number between 0 and 255. A setting of 255 will print approximately 1000 characters, about one screen full, per second. A speed setting of zero, on the other hand, will result in the printing of about 6 characters per second. The speed settings are not linearly proportional; thus determining approximate alternative settings will require some trial and error. When the computer is first turned on, SPEED is automatically set at 255, which is considered to be normal speed. To change the speed, say to 200, simply

type the command SPEED = 200. If desired, one may change the speed during the run of a program by inserting a statement such as

```
100  SPEED= 175
```

at the appropriate point in the program. A short example follows:

```
10  PRINT "THIS LINE PRINTS FAST"
20  SPEED= 50
30  PRINT "THIS LINE PRINTS SLOW"
40  SPEED= 255
```

A brief line by line description follows:

10 Prints the phrase at normal speed
20 Changes the speed to 50
30 Prints the phrase at the speed set in Line 20
40 Sets the speed back to normal

Because normal speed is 255, the speed set when the computer is first turned on, Line 10 will be displayed in approximately 0.02 seconds. Line 20 changes the rate at which characters are printed on the screen. Line 40 is an important line; its absence would keep the speed set at 50, even if a new program were loaded and run. The NEW command used before typing in a new program does not reset the speed setting. Therefore, when you change the speed in a program, include a statement that resets the speed to 255 before completing the program.

You need to experiment to determine what speed is best suited for certain segments of educational programs. The brief routine listed next allows you to experiment with the speed number without continually changing lines or rerunning the program.

```
10  INPUT "WHAT SPEED ";S
20  SPEED= S
30  PRINT "A QUICK BROWN FOX JUMPS OVER A LAZY DOG"
40  SPEED= 255
50  INPUT "TRY AGAIN (Y/N) ";Y$
60  IF Y$ = "Y" THEN  HOME : GOTO 10
70  END
```

Enter this program into your computer and run it several times, changing the input speed value. You may begin with zero and then go on to speeds that increment by 25: 0, 25, 50, . . . 225, 250. Find values that you feel create comfortable reading speeds. Try numbers

out of the valid range of 0 to 255. Try decimals! Once you've familiarized yourself with the various speed settings, you're ready to incorporate this feature into an educational program. Although the previous example is a utility program, it does contain programming techniques that should be included whenever the speed function is used.

10 Asks for the speed setting and assigns it to the variable S.
20 Sets the speed entered in Line 10.
30 Prints the phrase at the desired speed.
40 Resets the speed to 255.
50 Asks the user if another run is desired and assigns that response to the string variable Y$.
60 If another run is requested, then the screen is cleared and program execution returns to Line 10.
70 The program ends if the user does not respond with a Y.

Note that the speed was restored to 255 in Line 40. This change allows the TRY AGAIN message to appear instantaneously (0.016 seconds). It also guarantees restoration of normal speed should the user opt to end the program. User control messages discussed in Section 4.2 should not be printed at slower speeds, because the user generally anticipates such a message, and these speeds deny immediate access to the following frame since the message must appear on the monitor before a response can be accepted.

The speed function allows the programmer to emphasize important information and enables the user to read this information as it is being printed. It is wise to limit its use for presentation of only the most relevant information, thus preserving its effectiveness. Maintaining consistency in speed changes is also appreciated by the user. Continually altering text speed with different values produces inconsistent program flow. As you experiment with your program, you will eventually recognize optimum speed settings for your intended audience.

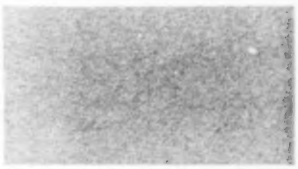
ON LINE 6.3

1 Explain how various SPEED settings can enhance an educational computer program in your teaching field.

2 Write a routine that requests the user to enter a letter. If a vowel is entered, the message YOU TYPED A VOWEL appears on the screen at a speed of 50. If a consonant is

entered, the message YOU ENTERED A CONSONANT appears at the normal speed of 255.

3 Discuss the similarities and differences between the SPEED command and the delay loop technique discussed in Section 3.4.



Random Function 6.4

The computer's ability to select numbers randomly can be most beneficial to the creator of educational programs. The Apple's random number generator returns a random real number that is greater than or equal to 0 and less than 1. To generate a random number in the immediate execution mode, simply type PRINT RND(1). This command will produce a number between 0.0000000000 and 0.999999999. Use of random numbers in an educational program usually calls for an integer, not a decimal fraction. To randomly generate an integer in a program, one would incorporate the INT function along with the RND function in a statement such as Line 10, which will generate an integer between 1 and 10, inclusive.

```
10  PRINT  INT ( RND (1) * 10 + 1)
```

RND(1) generates a random decimal between 0 and 1. This value is then multiplied by 10, resulting in a number between 0.000000000 and 9.99999999. After 1 is added, the new range becomes 1.000000000 to 10.99999999. Finally, the integer function INT, placed outside the parentheses, truncates the decimal, producing a whole number from 1 to 10 inclusive.

The general form for generating a random integer between any two whole numbers is

```
INT ( RND (1) * (H - L + 1) + L)
```

where H is the highest number in the range of integers to be selected and L is the lowest number in the range. If you wanted the computer to randomly generate an integer between 5 and 17 inclusive, then H would be 17 and L would be 5. The code to select and print this outcome is

```
10  PRINT  INT ( RND (1) * (17 - 5 + 1) + 5)
```

Simplified, the line becomes

```
10  PRINT  INT ( RND (1) * 13 + 5)
```

A random number generator has many uses in an educational program. One such use can be found in the following program that asks for the sum of two randomly generated integers.

```
95  HOME
100 X =  INT ( RND (1) * 9 + 1)
110 Y =  INT ( RND (1) * 9 + 1)
115  PRINT "WHAT IS THE SUM?"
120  VTAB 4: HTAB 15: PRINT X;" + ";Y;" = ";
130  INPUT A
140  IF A = X + Y THEN  PRINT  TAB( 16);"CORRECT": GOTO 160
150  PRINT  TAB( 5);"SORRY, THE CORRECT ANSWER IS ";X + Y
160  VTAB 24: PRINT "TRY ANOTHER PROBLEM (Y/N) ";
170  GET Y$
180  IF Y$ = "Y" THEN 95
```

A brief line by line description of this drill and practice program illustrates how the random function can be used to generate a variety of questions.

100 Generates a random integer between 1 and 9, inclusive, and sets it equal to the variable X.
110 Generates another random integer between 1 and 9, inclusive, and sets it equal to the variable Y.
115 Prints question.
120 Positions the output on the screen and prints the two random integers separated by a plus sign.
130 Allows the user to enter an answer.
140 If the answer entered is correct, it will be acknowledged.
150 An incorrect response will return a message and the correct solution.
160 The user is asked if another problem is desired.
170 Allows the user to reply to the request.
180 If a Y is entered, the program is rerun.

Use of the random function is not limited to numeric output. Words and phrases stored in string arrays can also be randomly selected. If random positive reinforcements are desired in response to correct answers, simply create a routine like this one:

```
10  FOR I = 1 TO 4
20  READ A$(I)
```

```
30  NEXT I
40  DATA GOOD JOB,THAT'S IT,RIGHT ON,CORRECT
50 Z =  INT ( RND (1) * 4 + 1)
60  PRINT A$(Z)
```

This routine will randomly print one of four remarks found in Line 40. An examination of the program lines reveals the ease with which words and phrases can be randomly selected.

10 Counter in the FOR/NEXT loop. In this case four iterations will be performed.
20 Reads a data element and stores it in array A$(I).
30 Returns to Line 10 to read another data element.
40 List of data to be read into the array.
50 Selects a random integer between 1 and 4, inclusive, and sets it to the variable Z.
60 Randomly prints one of the four phrases stored in the array.

Lines 50 and 60 may be executed as often as needed to produce positive reinforcements. There is no need to rerun Lines 10-30 however, because the data stored in the array will remain, unless you intentionally change it. Rerunning the FOR/NEXT loop that reads the data may very well result in a programming error.

After mastering the technique of randomly selecting words or phrases, you might want to prevent direct repetition in the printing of the words or phrases used in a program. For example, in the preceding routine CORRECT may be selected several times in succession during the program run. Avoiding repetition in the selection of random numbers can insure that all responses coded in the program will be used. The following program illustrates this programming technique. It prints four numbers randomly and without repetition.

```
100  FOR I = 1 TO 4
110 N(I) =  INT ( RND (1) * 4 + 1)
120  FOR J = 0 TO I - 1
130  IF N(I) = N(J) THEN 110
140  NEXT J
150  NEXT I
```

You can verify the selection of four different random numbers by entering the program in your computer, adding the following lines, and running the program several times. Although the numbers are selected randomly, no number is printed more than once.

```
160  FOR I = 1 TO 4
170  PRINT N(I)
180  NEXT I
```

How does the random selection without repetition routine work? First, Line 110 generates and stores a random integer in N(1). Then the first loop in Lines 120–140 checks to determine if the number has already been assigned to any of the previous array variables. Because N(1) is the first variable to be used, the inner loop (Lines 120–140) will not find any duplication, and program flow drops to Line 150. Line 150 transfers execution back to Line 100, where I is incremented to 2. Line 110 once again generates a random integer and now assigns it to the variable N(2). Lines 120–140 check all previously assigned array variables. If a duplication is found, control returns to Line 110, where another value is assigned to N(2). Once the value of N(2) successfully passes through the inner loop, I is incremented to 3, and the process is repeated until all array variables contain different numbers.

The loop that checks for duplication (Lines 120–140) starts with N(0). Starting J with 0 rather than 1 prevents the first element from being compared to itself, thus avoiding the creation of a futile infinite loop.

Once the numbers are selected, they can be used to generate random responses without repetition. The following program uses the random selection routine to select and print four different responses in a random order.

```
100  FOR I = 1 TO 4
110 N(I) =  INT ( RND (1) * 4 + 1)
120  FOR J = 0 TO I - 1
130  IF N(I) = N(J) THEN 110
140  NEXT J
145  READ A$(N(I))
150  NEXT I
155  DATA GOOD JOB,RIGHT ON,THAT'S IT,CORRECT
160  FOR I = 1 TO 4
170  PRINT A$(I)
180  NEXT I
```

Lines 145 and 155 are inserted into the random number routine, and the variable in Line 170 is changed to accept a string variable. Once the random number is selected and accepted by successfully passing through the loop in Lines 120–140, the first string in the DATA statement (Line 155) is read in Line 145. The string is assigned to the variable A$(N(1)), where N(1) is any random number from 1

to 4 inclusive. Line 150 then returns control to Line 100, where I is incremented to 2 and a second number is selected (Line 110). After completing the duplication check in Lines 120–140, the number is used to position a second data string into the array. This process continues until all the strings in the DATA statement have been assigned to the array.

This routine is valuable because duplication is prevented. The student user of a program can remain enthused and stimulated for a longer period of time if needless repetition is avoided. Just as instructors elicit the same information from a variety of questions, so too can the computer be programmed to test the same objective using similar, albeit different, questions.

We recommend that you study and practice the routine discussed here; change the data and number of elements. The test of mastery is its inclusion in a program of your own, perhaps one that asks random questions from a bank of 20, or a program that selects a U.S. president and asks his political affiliation. How about a program that requires the user to categorize words that contain either a long *O* sound or a short *O* sound! Whatever the topic, it is important not only to randomize but to prevent needless repetition during the run of an educational program.

ON LINE 6.4

1 List three uses of the random function in an educational program.

2 Write a program line that prints a random integer between 1 and 15.

3 Write a program line that prints a random integer between 5 and 20.

4 Write a program that randomly selects and prints an even positive integer less than 100.

5 Write a brief routine that randomly selects one of six colors from a DATA statement and prints that color on the screen. If the user wishes to run the routine a second time, have the routine reject the selection if the same color is selected.

Figure 6.1
In this routine five different random numbers are stored in the subscripted variable HB(I). The random numbers, ranging from 1 through 5, are used in the program to select animal heartbeat rates.

```
740  REM  GENERATE 5 DIFFERENT RANDOM NUMBERS FOR SELECTION OF AN ANIMAL
     AND STORE IN AN ARRAY
750  FOR I = 1 TO 5
760 RN =  INT (5 *  RND (1) + 1)
770 HB(I) = RN
780  FOR J = 0 TO I - 1
790  IF HB(I) = HB(J) THEN 760
800  NEXT J
810  NEXT I
```

6.5 Sorting Routines

A sorting routine can be used to rearrange a set of numbers in order from lowest to highest or highest to lowest. One example of the use of a sorting routine is a program in which classroom numbers are sorted and printed in a column, lowest to highest, as follows:

Sorted Room Numbers

235
267
269
283
325
345
348

The following short program sorts and prints a set of numbers. The user inputs the number of numbers in the set to be sorted, and this value is assigned to N (Line 20). As the FOR/NEXT loop in Lines 30–50 runs, the user inputs the numbers in the set, one by one, and they are stored in the subscripted variable S(I) in Line 40. The two FOR/NEXT loops (Lines 60–100) sort the set into a sequential order of lowest to highest numbers. Lines 110–130 comprise a FOR/NEXT loop that prints the rearranged set of numbers in this sequence.

```
10  PRINT "HOW MANY NUMBERS TO BE SORTED?"
20  INPUT N
25  DIM S(N)
30  FOR I = 1 TO N
32  PRINT "TYPE NUMBER ";I;" THEN PRESS RETURN."
40  INPUT S(I)
50  NEXT I
60  FOR I = 1 TO N - 1
70  FOR J = I + 1 TO N
80  IF S(I) > S(J) THEN T = S(I):S(I) = S(J):S(J) = T
90  NEXT J
100  NEXT I
110  FOR I = 1 TO N
120  PRINT S(I)
130  NEXT I

]RUN
HOW MANY NUMBERS TO BE SORTED?
?5
TYPE NUMBER 1 THEN PRESS RETURN.
?450
TYPE NUMBER 2 THEN PRESS RETURN.
?186
TYPE NUMBER 3 THEN PRESS RETURN.
?225
TYPE NUMBER 4 THEN PRESS RETURN.
?302
TYPE NUMBER 5 THEN PRESS RETURN.
?288
186
225
288
302
450
```

Lines 60–100 contain two FOR/NEXT loops, one nested inside the other, that constitute the heart of the program. It is here that the numbers stored in the subscripted variable S(I) are sorted into an ordered sequence. Line 80 constitutes a "switching routine" that orders the numbers. When Line 80 executes, a number, S(I), is compared with the next number in the set S(J). If S(I) is larger than S(J), the IF condition is met, and the value of S(I) is assigned to T. S(I) is then assigned the value of S(J); S(J) is assigned the value of T. Thus the values of S(I) and S(J) are exchanged, and S(J) is now a larger number than S(I). In effect, T serves as a value holder during the exchange between S(I) and S(J).

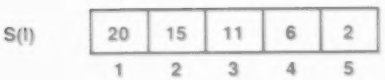
The values of S(I) and S(J) are determined by the subscript numbers specified by the counters in Line 60 (FOR I = 1 TO N – 1)

and Line 70 (FOR J = I + 1 TO N). Table 6.1 lists the sequence of subscript numbers produced during a run in which a set of five numbers (N = 5) is sorted.

Table 6.1
Values of I and J when five numbers are sorted

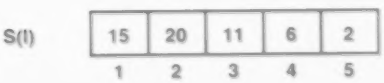
I (1 TO N − 1)	J (I + 1 TO N)
1	2,3,4,5
2	3,4,5
3	4,5
4	5

To understand the sorting routine, consider a specific set of five numbers stored in the subscripted variable S(I), as shown:

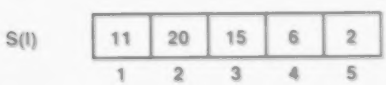


S(1) = 20 S(2) = 15 S(3) = 11 S(4) = 6 S(5) = 2

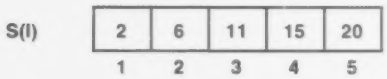
The first time through the outer FOR/NEXT loop (Lines 60–100), I has the value of 1 and J has the value of I + 1, which is 2. Therefore, S(1) is compared against S(2) in Line 80. Since S(1) with a value of 20 is greater than S(2) with a value of 15, Line 80 is executed and the two numbers are exchanged. The sequence of numbers at this point is



During the second iteration of the inner loop (Lines 70–90), I continues to have a value of 1. However, the value of J is incremented to 3 by the counter in Line 70. S(1) is now compared with S(3). Since 15 is greater than 11, the values of S(1) and S(3) are switched to form the sequence



As the iterations of the sorting routine proceed, the subscripts of the variables are increased (Table 6.1), and the values of the respective subscripted variables S(I) and S(J) continue to be compared and sorted. When the iterations of the loop are completed, the set of numbers has been sorted into the ordered sequence



A sorting routine can also be used to alphabetize words. Study the following program and note that it is similar to the number sorting routine discussed earlier. However, the alphabetizing routine is modified to use the subscripted string variable S$(I) in place of the subscripted numeric variable S(I) used in the number sorting routine. If a set of names is entered when this program is run, the names are sorted and printed in alphabetical order.

```
10  PRINT "HOW MANY NAMES TO BE ALPHABETIZED?"
20  INPUT N
25  DIM S$(N)
30  FOR I = 1 TO N
32  PRINT "TYPE NAME ";I;" THEN PRESS RETURN."
40  INPUT S$(I)
50  NEXT I
60  FOR I = 1 TO N - 1
70  FOR J = I + 1 TO N
80  IF S$(I) > S$(J) THEN T$ = S$(J):S$(J) = S$(I):S$(I) = T$
90  NEXT J
100  NEXT I
110  FOR I = 1 TO N
120  PRINT S$(I)
130  NEXT I


]RUN
HOW MANY NAMES TO BE ALPHABETIZED?
?5
TYPE NAME 1 THEN PRESS RETURN.
?CARL
TYPE NAME 2 THEN PRESS RETURN.
?BETSY
TYPE NAME 3 THEN PRESS RETURN.
?DAN
TYPE NAME 4 THEN PRESS RETURN.
?JANE
TYPE NAME 5 THEN PRESS RETURN.
?HENRI
```

```
BETSY
CARL
DAN
HENRI
JANE
```

In Line 80 the string values of the names stored in S$(I) are compared. This comparison is based on the ASCII values of the letters in the names. ASCII values for letters increase in accord with the order of the letters in the alphabet. ASCII values increase numerically from 65 for A to 90 for Z. See Table 6.2.

Table 6.2
ASCII values of letters

A	65	H	72	O	79	U	85
B	66	I	73	P	80	V	86
C	67	J	74	Q	81	W	87
D	68	K	75	R	82	X	88
E	69	L	76	S	83	Y	89
F	70	M	77	T	84	Z	90
G	71	N	78				

Consider the five names CARL, BETSY, DAN, JANE, HENRI stored in the subscripted variable S$(I) as follows:

S$(I)	CARL	BETSY	DAN	JANE	HENRI
	1	2	3	4	5

When the string values of names are compared in Line 80, the names are sorted alphabetically. For instance, when CARL (S$(1)) is compared with BETSY (S$(2)), the higher ASCII value of CARL causes the names to be exchanged between the variables (S$(1) > S$(2)) so that BETSY comes before CARL. Repeated comparisons on this basis rearrange the words stored in S$(I) into alphabetical order.

S$(I)	BETSY	CARL	DAN	HENRI	JANE
	1	2	3	4	5

You can use a program such as the following to sort names and corresponding numbers. For example, a set of names of students

and their corresponding room numbers may be sorted and printed in order from lowest to highest, as shown in the run that follows the listing.

```
6  REM  NN=NUMBER OF NAMES :N$=NAME:NU=NUMBER
20  PRINT "HOW MANY NAMES?"
30  PRINT "TYPE THE NUMBER.  THEN PRESS RETURN."
40  INPUT NN
42  DIM N$(NN)
43  DIM NU(NN)
50  FOR I = 1 TO NN
60  PRINT "TYPE NAME ";I;" THEN PRESS RETURN."
65  INPUT N$(I)
70  PRINT "TYPE THE ROOM NUMBER - THEN PRESS RETURN"
75  INPUT NU(I)
80  NEXT I
100  FOR I = 1 TO NN - 1
110  FOR J = I + 1 TO NN
120  IF NU(I) < NU(J) THEN 190
130 T = NU(I)
135 NU(I) = NU(J)
140 NU(J) = T
145 T$ = N$(I)
150 N$(I) = N$(J)
160 N$(J) = T$
190  NEXT J
195  NEXT I
200  FOR I = 1 TO NN
210  PRINT N$(I),
220  PRINT NU(I)
230  NEXT I

]RUN
HOW MANY NAMES?
TYPE THE NUMBER.  THEN PRESS RETURN.
?6
TYPE NAME 1 THEN PRESS RETURN.
?SUSAN
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?285
TYPE NAME 2 THEN PRESS RETURN.
?CHRIS
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?322
TYPE NAME 3 THEN PRESS RETURN.
?BILL
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?150
TYPE NAME 4 THEN PRESS RETURN.
?HELEN
```

```
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?401
TYPE NAME 5 THEN PRESS RETURN.
?HENRI
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?385
TYPE NAME 6 THEN PRESS RETURN.
?LUIS
TYPE THE ROOM NUMBER - THEN PRESS RETURN
?325
BILL                    150
SUSAN                   285
CHRIS                   322
LUIS                    325
HENRI                   385
HELEN                   401
```

Recalling the earlier explanation of sorting routines, you should be able to interpret the listing of the program and understand how the sorting of the names parallels the sorting of the numbers so that the names and numbers continue to correspond in the printed output. Note in Line 120 that the sorting depends on the relative sizes of the numbers. If NU(I) is greater than NU(J), the numbers stored in the two variables are switched and the corresponding names stored in N$(I) and N$(J) are likewise switched. If NU(I) is less than NU(J), there is no switching and change in the order because the program goes to Line 190 and bypasses Lines 120–160, in which numbers and names are sorted.

Words and corresponding numbers can be sorted so that the words are printed in alphabetical order along with the corresponding numbers. A program that does this alphabetizing is similar to the preceding program. However, Lines 120–160 are changed as follows:

```
120  IF N$(I) < N$(J) THEN 190
130 T$ = N$(I)
135 N$(I) = N$(J)
140 N$(J) = T$
145 T = NU(I)
150 NU(I) = NU(J)
160 NU(J) = T
```

This change causes the sorting to be based on the string values of the names stored in N$(I). The sorting of the numbers parallels the alphabetical sorting of the names, resulting in the alphabetized list of names and room numbers shown in Figure 6.2.

```
BILL      150
CHRIS     322
HELEN     401
HENRI     385
LUIS      325
SUSAN     285
```

Figure 6.2
Alphabetized listing of students and room numbers.

If you are unfamiliar with computer sorting, you should enter the programs presented here and run them. Note and compare the formats of the output for each program and save the programs on a diskette for future reference and use. If your educational computer program involves sorting names or numbers, you can use one or more of the routines presented here. Be certain to include prompts that inform the student about the kind of string inputs required, like names, and the corresponding numbers to be inputted such as dimensions, averages, or scores.

ON LINE 6.5

1 How is a sorting routine for strings different from a sorting routine for numbers?

2 List five examples of educational programs in which a sorting routine could be used. Explain the purpose of the sorting routine in each program.

3 Write a program for a science class in which a student enters the names of five elements and the density of each element. The program should then print the names of the elements and their densities in order from lowest to highest density.

4 Write a program in which the user enters the names of five baseball teams, entertainers, or popular songs. The user is then directed to rate the popularity of each entry on a scale of 1 through 5. The program should print the entries in order, based on the rated popularity.

Digital Counter 6.6

A digital counter is a device that uses numbers (digits) to count. For example, an odometer on an automobile dashboard is a device that

counts the number of miles traveled. You can easily create a digital counter on your computer screen by programming these six lines and running the routine.

```
20  FOR I = 1 TO 100
40  VTAB 11: HTAB 20
50  PRINT I
60 SOUND =  PEEK ( - 16336):SOUND =  PEEK ( - 16336)
70  FOR P = 1 TO 400: NEXT P
80  NEXT I
```

When the program is run, numbers from 1 to 100 are printed sequentially in the same spot on the screen, each number replacing the previous number. Also, a click sound is heard each time a number is printed.

The digital counter program is based on a FOR/NEXT loop (Lines 20–80) in which the value of the variable I in Line 20 is incremented from 1 to 100. In Line 50 the PRINT I statement prints a number on the screen that is the value of I at the time Line 50 is executed. The first time through the loop a 1 is printed, the second time a 2 is printed, the third time a 3 is printed, and so on, up to 100. Line 40 sets the row and column coordinates for printing the numbers. Execution of Line 60 produces the click sound that accompanies the printing of each number.

Memory location −16336 is connected to the speaker in such a way that when it is read with a PEEK function (Line 60), the speaker makes a click. The second statement in Line 60 resets the speaker for another click the next time the memory location is PEEKed. Only one click is heard each time Line 60 is executed.

Line 70 contains a delay loop that produces a short pause between the printing of each successive number. You can alter the rate at which the counter counts by changing the statement in Line 70. Increase the higher number in the FOR statement to slow the counter, and, conversely, decrease the number to speed up the counting. Increasing the number increases the pause time and slows the counting. Decreasing the number decreases the pause time and speeds up the counting.

You can change the counter so that it counts to whatever number you choose. Simply change the FOR statement in Line 20 to the number desired. For example, if you want the routine to count to 60 change Line 20 to FOR I = 1 to 60.

An enhancement of the counter may be made so that it ticks off seconds and tenths of a second as a digital timer. When the program runs, the timer prints a decimal point on the screen and digits

to the right and left of the decimal point. The digits to the right show tenths of a second. Digits to the left indicate seconds. For example, nine tenths of a second appears as 0.9; thirty-nine and eight-tenths seconds is printed as 39.8. Here is a program for a 60 second digital timer.

```
15  VTAB 11: HTAB 20
17  PRINT "."
20  FOR I = 0 TO 60
22  VTAB 11: HTAB 19
23  IF I > 9 THEN  VTAB 11: HTAB 18
25  PRINT I
30  FOR J = 0 TO 9
40  VTAB 11: HTAB 21
42  PRINT J
50 SOUND =  PEEK ( - 16336):SOUND =  PEEK ( - 16336)
55  FOR P = 1 TO 49: NEXT P
60  NEXT J
100  NEXT I
```

The digital timer is calibrated in seconds by the delay loop in Line 55. Statements in this line, FOR P = 1 TO 49 : NEXT P, produce the precise delay needed in the run of the loop so that there is a 1 second interval between each printing of the numbers that count the seconds of time. You can change the length of time the timer runs by changing the counting number in the FOR statement (Line 20). For example, by changing the statement to FOR I = 0 TO 10, the timer will run for approximately 10 seconds.

When the timer is running, the outer loop in the routine prints seconds, and the inner loop prints tenths of a second. Variable I is used to print the number of seconds in accord with the value assigned by the FOR statement in Line 20. Tenths of a second are printed by executing the statement PRINT J in Line 42. The number to be printed is assigned to J by the FOR statement in Line 30. The following specific explanation of the coding of certain lines in the routine will help you understand how the routine executes.

15 Moves the cursor to the specified point on the screen for printing the decimal point

22–23 Moves the cursor to the left of the decimal point for the printing of seconds

40 Moves the cursor to the right of the decimal for the printing of tenths of a second

You can change the programming of a digital timer so that it runs to a random number of seconds. Add a line that will generate a random integer within a certain range, such as Line 12, which generates a random number from 1 to 60. Then change Line 20 so that it counts from 0 to N, the variable for the random number.

```
12 N =  INT (60 *  RND (1) + 1)
20  FOR I = 0 TO N
```

One example of the use of a digital timer is a program named Blastoff, in which a rocket is printed on the screen and then moved up and off the screen to simulate a rocket firing and liftoff. The digital timer printed on the screen indicates the number of seconds the rocket is in flight. In using this program, the student calculates the "height" of the rocket based on the time the rocket is in flight.

A timer could also be used in a biology program in which students calculate the rate of human respiration or the rate of human heartbeat. If your program involves subject matter related to counting, timing, or calibration of instruments, you may find an opportunity to include a counter or digital timer by using the routines described here.

ON LINE 6.6

1 Input the digital counter program on page 180 into your computer and run it. Using a watch with a second hand, observe the time it takes the computer to count to 100.

2 Write a program that uses a digital counter to teach counting by 10s.

3 Input the 60 second digital timer into your computer and run the program. Using a watch with a second hand, check the time it takes to count to 60 seconds. If necessary, modify the delay loop in Line 55 and recalibrate the program so that it counts to 60 in 60 seconds.

4 Write a program that uses a digital timer to teach the rate of human heartbeat, respiration, or other repetitive phenomena.

5 Change the digital counter routine so that it counts down from 10 to 0 and prints BLASTOFF when it reaches 0.

Character Graphics 6.7

An easy way to add interest and increase the meaningfulness of an educational program is through the use of character graphics in text mode. Character graphics, as the term implies, refers to the printing of figures or pictures created with letters from the alphabet and keyboard characters such as <, #, *, -, and /. For example, simple figures such as a square or a triangle for a geometry lesson may be created with the letter X. More complex graphics figures, such as a bird, automobile, or spaceship, may be developed by printing selected keyboard characters at various points on the screen so that they collectively combine to form a picture. Examples of character graphics in an educational program are a rocket, ruler, and arrow. Each of these figures is used to help illustrate a fact or principle: the rocket to simulate the motion of a space vehicle, the ruler to show units of metric measure, and the arrow to represent a force that moves a wheelbarrow.

Here is a short program that prints an arrow made of two keyboard characters (- and >). The arrow could be used in a program to point to important parts of each frame.

```
5  HOME
10  VTAB 12
20  HTAB 17
30  PRINT "------>"
```

You may wish to experiment with the elementary use of character graphics by entering this program into your computer, running it, and changing the VTAB and HTAB values so that the arrow is printed at different places on the screen. Although quite simple in its coding, the arrow printing program illustrates the two basic principles of character graphics programming: (1) appropriate keyboard characters must be selected to create the graphic; and (2) TAB functions (TAB, HTAB, VTAB) must be used to position the characters on the screen so that they combine to form the desired figure.

Here is a program that moves character graphics programming a step up from the arrow program. Enter it into your computer, save it on diskette, and then run the program.

```
5  HOME
7  VTAB 5
10  PRINT  TAB( 18)"!!!!!!!"
```

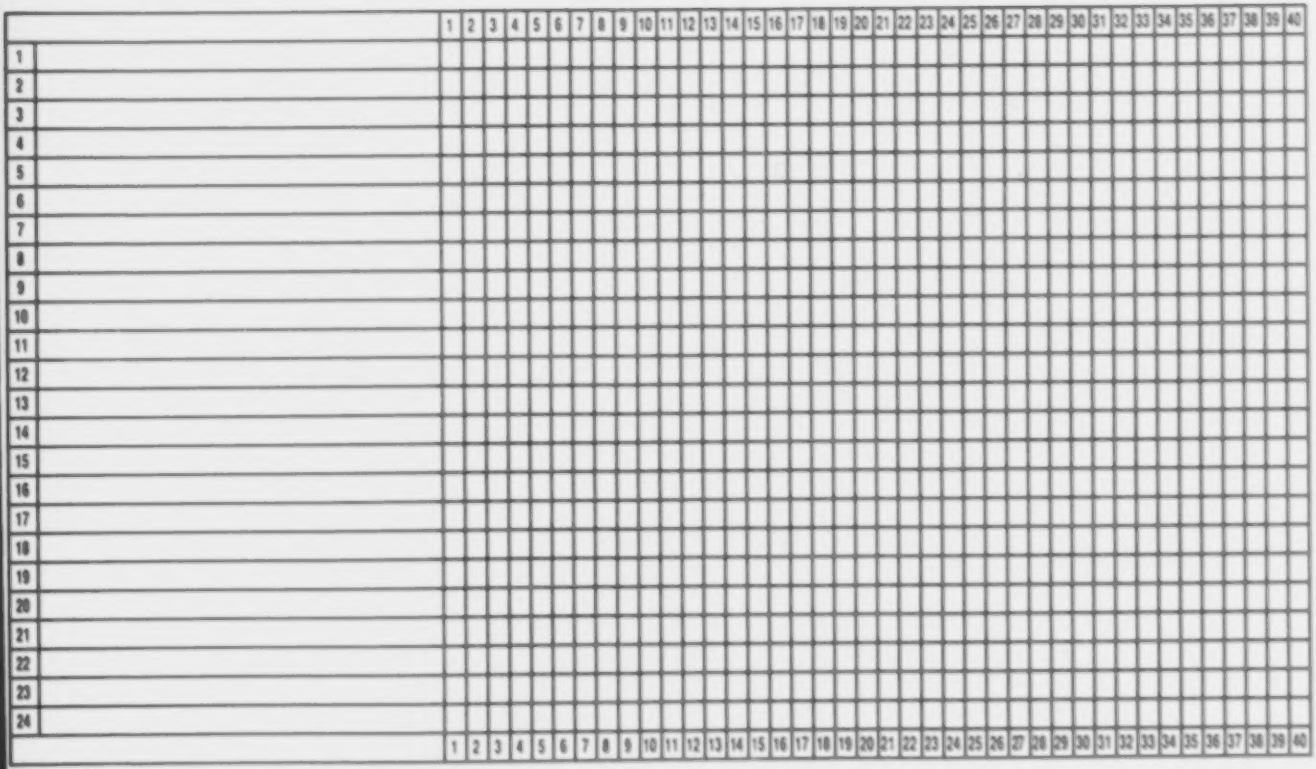
```
20  PRINT  TAB( 15)".............."
30  PRINT  TAB( 14)".                .."
40  PRINT  TAB( 13)".                    .."
60  PRINT  TAB( 12)".      0       0       .."
70  PRINT  TAB( 12)".                       .."
80  PRINT  TAB( 10)"((.                        .))"
90  PRINT  TAB( 12)".          XX           .."
100  PRINT  TAB( 12)".                       .."
110  PRINT  TAB( 13)".      #       #      .."
120  PRINT  TAB( 14)".      #    #      .."
130  PRINT  TAB( 15)".      ###      .."
135  PRINT  TAB( 16)".              .."
150  PRINT  TAB( 17)"........."
```

```
]RUN
```

Have you seen a happier face recently? Note that it is made up of seven different keyboard characters. These characters are positioned on the screen by the TABs in the PRINT statements. VTAB 5 in Line 7 sets the printing of the top of the face five lines down from the top of the screen, thereby positioning the picture near the center of the screen. Remember, as with other routines, you can put the happy face or any other kind of character graphic in a subroutine and print it on the screen by calling it with a GOSUB statement at any point in your program. For younger students the happy face could be printed as a positive reinforcement whenever a correct answer to a question is entered.

Plan the character graphic figure on a program planning grid like the one shown in Figure 6.3. The grid has a box for each column/row coordinate on the Apple screen, which is 40 columns wide and 24 columns high. First, draw the graphic on the grid by marking the keyboard characters you will use in the boxes that correspond

Figure 6.3
Program planning and output grid.



to the positions on the screen where the characters will be printed. Then in the space to the left write the line number and PRINT statement that corresponds to each row of characters. Create the program by typing these lines into your computer. As you gain experience in creating character graphics through this method, you will quickly see the correspondence between the planning grid, line statements, and screen output and will easily be able to edit the line statements as needed so that the graphic is printed in exact accordance with your specifications. Your character graphics need not be limited to stationary figures. A graphic character may be animated so that the whole figure or part of it appears to move. This is done by adding lines of code that cause the figure to be printed successively in different positions on the screen. One way to do this is to print the figure on the screen, clear the screen with a HOME statement, and then print the figure again at a position directly adjacent to where it was. This technique is exemplified in this next program, which prints and moves the arrow across the screen.

```
5  FOR I = 1 TO 34
10  HOME
20  VTAB 12
25  HTAB I
30  PRINT "------>"
40  NEXT I
```

```
1990  REM  CHARACTER GRAPHIC OF THE NUMBER MACHINE
2000  VTAB 2: HTAB 10: PRINT "/////////////////////////////"
2010  VTAB 3: HTAB 9: PRINT "/////////////////////////////"
2020  VTAB 4: HTAB 8: PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!///"
2030  VTAB 5: HTAB 8: PRINT "!.............................///"
2040  VTAB 6: HTAB 8: PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!///"
2050  VTAB 7: HTAB 8: PRINT "!:.                          .:///"
2060  VTAB 8: HTAB 8: PRINT "!:.                          .:///"
2070  VTAB 9: HTAB 8: PRINT "!:.............................:///"
2080  VTAB 10: HTAB 8: PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!///"
2090  VTAB 11: HTAB 8: PRINT "!.............................///"
2100  VTAB 12: HTAB 8: PRINT "!.............................//"
2110  VTAB 13: HTAB 8: PRINT "!_____________________________/"
2120  RETURN
```

Figure 6.4
This subroutine in The Number Machine prints a box on the screen. A number entered by the student is printed in the center expressed literally in words.

Note Lines 10–30 in the FOR/NEXT loop. The first time through the loop, the arrow is printed by Line 30 at the left edge of the screen, as specified by HTAB I (I = 1). Each successive time through the loop the screen is cleared by the HOME statement in Line 10, and the arrow is printed again one space to the right of where it was printed before, because the value of I for HTAB I increments by 1 with each iteration of the loop. The repeated printing and clearing of the arrow in this way makes the arrow appear to travel right across the screen. Animation of a graphic arrow may also be produced without the screen clearing HOME statement by printing the arrow on the screen, clearing the arrow by printing spaces at its present position, and then printing the arrow again one space to the right. Here's how it is done.

```
2  HOME
5  FOR I = 1 TO 34
20  VTAB 12
25  HTAB I
30  PRINT "----->"
35  FOR P = 1 TO 25: NEXT P
40  VTAB 12
50  HTAB I
60  PRINT "      "
70  NEXT I
```

Observe that the HOME statement appears only in Line 2 and is executed only once during the run of the program. The arrow is printed at VTAB 12, HTAB I by Line 30, followed by a delay loop in Line 35 that gives a pause before the arrow is wiped out by the printing of spaces (Line 60). There are six spaces between the quotation marks in Line 60, which correspond to the six characters that make up the arrow. Consequently, execution of Line 60 clears the arrow from the screen. The repeated printing and clearing of the arrow produces its apparent motion across the screen.

Although the foregoing program does the job, a more efficient program for moving the arrow is:

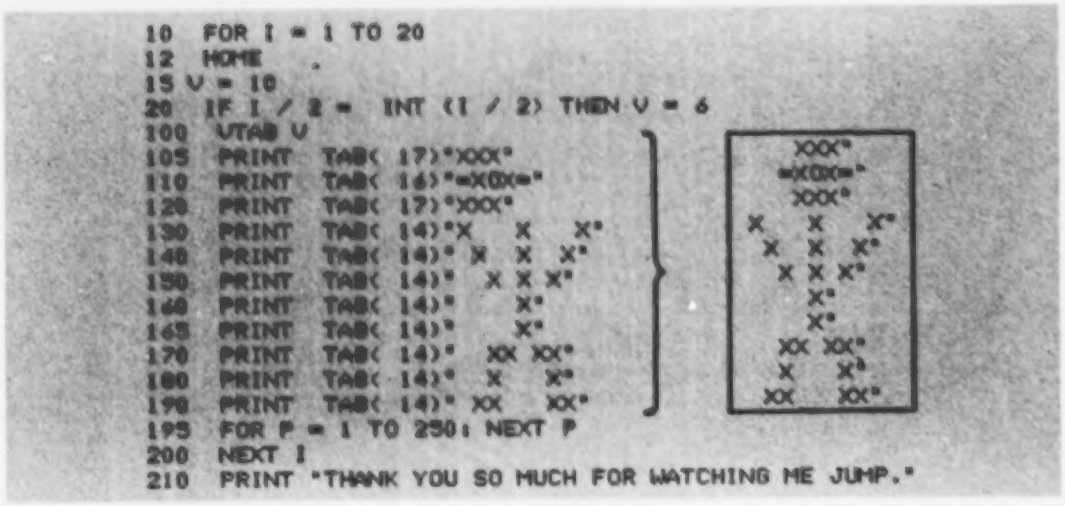
```
2  HOME
5  FOR I = 1 TO 34
20  VTAB 12
25  HTAB I
30  PRINT " ----->"
35  FOR P = 1 TO 25: NEXT P
40  NEXT I
```

Observe in Line 30 the space in the string to the left of the arrow. When the string is printed repeatedly as the loop iterates and I in-

crements from 1 to 34, the arrow appears after each printing, one space to the right of its former position. At the same time the space in the string wipes out the left end character (-) from the previous printing of the arrow. The effect, as in the arrow programs discussed earlier, is the arrow's apparent motion from left to right.

When programming the animation of a figure, the HOME statement causes any text on the screen to "flash" as the text is repeatedly cleared and reprinted. This unwanted output is avoided by printing one or more spaces to delete all or a portion of the graphics figure as it is reprinted.

One more character graphics program, slightly more imaginative than the moving arrow, is given in Figure 6.5. When the program is run, a jumping "space critter" is displayed on the screen.



```
10  FOR I = 1 TO 20
12  HOME
15 V = 10
20  IF I / 2 =  INT (I / 2) THEN V = 6
100  VTAB V
105  PRINT  TAB( 17)"XXX"
110  PRINT  TAB( 16)"=XOX="
120  PRINT  TAB( 17)"XXX"
130  PRINT  TAB( 14)"X   X   X"
140  PRINT  TAB( 14)" X  X  X"
150  PRINT  TAB( 14)"  X X X"
160  PRINT  TAB( 14)"    X"
165  PRINT  TAB( 14)"    X"
170  PRINT  TAB( 14)"  XX XX"
180  PRINT  TAB( 14)"  X   X"
190  PRINT  TAB( 14)" XX   XX"
195  FOR P = 1 TO 250: NEXT P
200  NEXT I
210  PRINT "THANK YOU SO MUCH FOR WATCHING ME JUMP."
```

Figure 6.5
Space critter graphic.

Note that the lines for printing the space critter are within the FOR/NEXT loop established by Lines 10-200. Each time the loop is executed, the screen is cleared by the HOME statement (Line 12), and the figure is printed at VTAB V. V is equal to either 10 (Line 15) or 6 (Line 20), depending on whether the value of I is odd or even. If the value of I is odd, the figure is printed at Row 10; if even, at Row 6. Each time through the loop the value of I as it increments from 1 through 20 is changed, odd to even or even to odd. Therefore, the printing of the critter is alternately raised to Row 6 and

lowered to Row 10, making it appear to jump up and down. Line 20 sets the condition for determining if I is an even number. I is even if I divided by 2 is equal to the integer of I divided by 2.

If you use your imagination, and select suitable keyboard characters, you will be pleasantly surprised to see how, in a relatively short time, you can add graphic figures to your program that enhance its vitality and interest. The techniques presented in this section will help you get started. As you gain experience in their use, you will no doubt move on to the development of more complex character graphics than those presented here. Giving some thought and time to the character graphics endeavor will quite likely result in your coming up with something imaginative and interesting that will contribute significantly to the visual impact of the program when it is run by a student.

---

1 Write a short program that prints an X in the center of the screen and moves the X five rows down the screen.

2 Change the program you wrote in question 1 by adding lines that cause the X to move randomly up, down, left, or right. (*Hint:* The program must contain a line that generates a random number from 1 through 4 and assigns the number to a variable.)

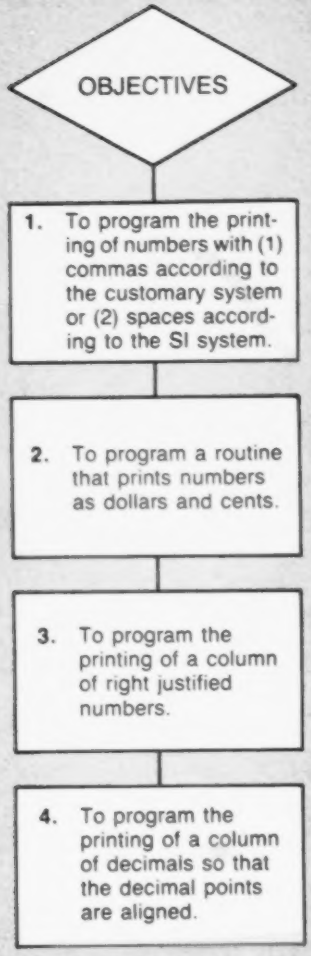
3 Make a photocopy of the planning grid on page 185. Use it to create a program that prints an animal face in character graphics. Use your imagination to design the graphic figure.

4 Write a program that prints a space critter on the computer screen.

5 Enhance your space critter program so that the whole critter or part of it moves by animation.

CHAPTER SEVEN

NUMERIC OUTPUT


OBJECTIVES
1. To program the printing of numbers with (1) commas according to the customary system or (2) spaces according to the SI system.
2. To program a routine that prints numbers as dollars and cents.
3. To program the printing of a column of right justified numbers.
4. To program the printing of a column of decimals so that the decimal points are aligned.

Programs involving numbers and calculations often require printing numeric output formatted so that it is easy to read and comprehend. This chapter begins with a routine that adds either commas or spaces to a number before it is printed. For example, commas added to the number 200000000 cause it to be printed on the computer screen as 200,000,000.

In the section on dollars and cents, directions are given for writing a routine that takes a calculated value such as 1.1 and prints it as $1.10. When printing a column of numbers, the numbers often are right justified, meaning that the rightmost digit of each number in the column is aligned vertically. A snappy routine that does this expeditiously is discussed in Section 7.4. The solution to another number formatting problem, printing columns of decimal numbers so that decimal points are aligned properly, is also explained.

7.1 Commas in Numerals

If you multiply 1000*1000 on your microcomputer, you will see the answer 1000000 printed on the screen. Although 1000000 is correct, the printing of the numeral differs from the style, commonly used in the United States, in which commas are used in numerals to demark thousands. For ease in reading, 1000000 is printed as 1,000,000. Fifty million is printed as 50,000,000.

In the International System of metric units (SI) a space is used instead of a comma to separate thousands. Table 7.1 compares the three forms of writing numerals.

Table 7.1
Comparative printing of numerals

Computer	*Commas Added*	*SI*
5000	5,000	5 000
777777	777,777	777 777
1736000	1,736,000	1 736 000

The following program contains a subroutine you can include in an educational computer program that adds commas to a number. On most computers it can be used for numbers up to nine digits. Numbers larger than this will be printed as exponents. In the main part of the program a number to be printed is calculated and assigned

to N$ as a string. Then the subroutine at Line 7000 is called, which adds commas to the number before it is printed.

```
10 N = 50 * 5000
20 N$ =  STR$ (N)
30  GOSUB 7000
40  PRINT Z$
6490  END
7000  FOR I =  LEN (N$) TO 1 STEP  - 1
7010 M$ =  MID$ (N$,I,1)
7020 C = C + 1
7030  IF C = 3 AND I <  > 1 THEN M$ = M$ + ",":C = 0
7040 NN$ = NN$ + M$
7045  NEXT I
7050  FOR I =  LEN (NN$) TO 1 STEP  - 1
7060 Z$ = Z$ +  MID$ (NN$,I,1)
7070  NEXT I
7080  RETURN
```

Here is an explanation of how the subroutine works. In Line 10, N is assigned a value of 250000, the product of 50 * 5000. The string value of N ("250000") is assigned to N$ in Line 20. Line 30 calls the subroutine at Line 7000. In the FOR/NEXT loop (Lines 7000–7045) the digits in N$ are concatenated along with the appropriate commas into NN$. As the loop iterates, each digit is assigned to M$ (Line 7010), from right to left, and the number of digits assigned is counted by Line 7020. After each set of three digits a comma is concatenated into the string NN$ (Lines 7030 and 7040). When the run of the FOR/NEXT loop is completed, the string "000,052" has been assigned to NN$. Note that the digits in NN$ are reversed from the order they hold in N$.

Lines 7050–7070 constitute a FOR/NEXT loop that reverses the order of the digits in NN$ and concatenates them into a string, "250,000", assigned to Z$ in Line 7060. After the completion of the iterations of this loop, the subroutine is completed and Line 7080 returns the program to Line 40, which prints the numeral as 250,000.

It is important to note in the subroutine that the digits in N$ are assigned to M$, from right to left, as the loop runs. It is done this way because it is necessary to count the digits from right to left so that a comma may be inserted after each set of three digits. Consequently, the digits and commas are concatenated into NN$ in reverse order. The loop that prints out the numeral must therefore begin with the digit at the far right of NN$ and print the digits in reverse order for the number to be printed correctly as 250,000.

Figure 7.1
This subroutine in The Number Machine adds commas to a number entered by a student.

```
4000  REM  ROUTINE TO INSERT COMMAS IN NUMBER
4010  FOR I =  LEN (N$) TO 1 STEP  - 1
4020  N$ =  MID$ (N$,I,1)
4030  C = C + 1
4040  IF I = 1 AND I <  > 1 THEN N$ = N$ + ",":C = 0:NC = NC + 1
4050  NW$ = N$ + NW$
4060  NEXT I
4070  FOR I =  LEN (NW$) TO 1 STEP  - 1
4080  Z$ = Z$ +  MID$ (NW$,I,1)
4090  NEXT I
4100  RETURN
```

Should you write a program in which one or more large numbers will be calculated and subsequently printed, you may see fit to include the Comma Insertion subroutine. For each number to be printed, write a statement that assigns to N$ the string value (STR$) of the number. Follow with a GOSUB statement that calls the subroutine and a PRINT statement that prints the numeral with the added commas.

If your program uses the SI metric system, modify the subroutine so that it adds a space instead of a comma. Do this by deleting the comma between the quotation marks in Line 7030; deleting the comma leaves a space between the quotation marks that will be inserted at the appropriate positions within the numeral.

ON LINE 7.1

1 Write a program that makes the indicated calculations and prints the answers:
   a. 500 * 500
   b. 1000000 – 500000
   c. 7500 + 11000

2 Add a subroutine to the program that adds commas to the numeric answers before they are printed.

Scientific Notation Conversion 7.2

Certain educational programs require use of numbers larger than 1 billion. Large numbers such as those invelved in the conversion of miles to inches, measurement of astronomical distances, or drill and practice in multiplying quantities like 1 million times 1 million may play havoc with your program. As with most microcomputers, Apple outputs large numbers in scientific notation, a topic that may be too advanced for the intended audience of your program. If one entered a number like 623,000,000,000 (623 billion) into the computer in the form of the command

PRINT 623000000000

the output, after pressing Return, would be 6.23E+11, certainly not a numeral that would comfort a student who lacked knowledge of scientific notation. The following subroutine converts any large number from scientific notation back to its original long form.

```
8000 N$ =  STR$ (N)
8010 EX =  VAL ( RIGHT$ (N$,2))
8020  FOR A = 1 TO 11
8030  IF  MID$ (N$,A,1) <  > "E" THEN  NEXT A
8040 M$ =  LEFT$ (N$,1)
8050  IF A = 2 THEN A = 3: GOTO 8090
8060  FOR X = 3 TO A - 1
8070 M$ = M$ +  MID$ (N$,X,1)
8080  NEXT X
8090  FOR X = 1 TO EX - (A - 3)
8100 M$ = M$ + "0"
8110  NEXT X
8120  RETURN
```

To understand the workings of this subroutine, enter it into your computer. Then add Lines 10–60. Run the program and experiment with a variety of large numbers.

```
10  INPUT "ENTER A NUMBER ";N
20  IF N < 1000000000 THEN  PRINT "ENTER A LARGER NUMBER": GOTO 10
30  PRINT "NUMBER BEFORE CONVERSION:"
35  PRINT N
40  GOSUB 8000
50  PRINT : PRINT "NUMBER AFTER CONVERSION:"
55  PRINT M$
60  END
```

The conversion subroutine beginning at Line 8000 changes the number entered to a string so that string functions can be used. For example, if 623000000000 is entered in Line 10, then the resulting string assigned to N$ in Line 8000 will be 6.23E+11. Line 8010 assigns the rightmost two digits (11) to the variable EX, representing the exponent. The loop in Lines 8020 and 8030 is used to locate the position of E. In our example of 6.23E+11, the E is in the fifth position, so a 5 is assigned to the variable A. Line 8040 assigns the first digit of N$, in this case 6, to the eventual final answer, which is represented by M$. Line 8050 is a conditional statement that omits execution of the loop in Lines 8060–8080 if the original number entered contains only one significant digit. For example, if the number entered in Line 10 is 2000000000, then program execution will transfer to Line 8090 since the number has only one significant digit (2). The loop in Lines 8060–8080 adds all remaining digits to M$. Starting at the third position, because the second position of the string contains the decimal point, this loop concatenates the remaining digits to M$. Since the E is in the position assigned to the variable A (Lines 8020–8030), this loop will end at position A-1. The concluding loop of this conversion subroutine (Lines 8090–8110) adds as many zeroes as indicated by the exponent.

This subroutine may prove to be very useful when numerical calculations are required. For instance, on the elementary level one may create a program whereby a student enters his or her age in years and the output displays the number of seconds that person has lived! You can modify the preceding program so that it makes this calculation. Change Lines 10 and 20 and add Line 16 as shown here:

```
10  INPUT "HOW OLD ARE YOU? ";A
13  REM CALCULATES AGE IN SECONDS
16 N = A * 365 * 24 * 60 * 60
20  IF N < 1000000000 THEN  PRINT N: END
```

Line 16 calculates the number of seconds based on the value in years entered in Line 10. The condition in Line 20 results in the printing of N since the number of seconds for years less than 32 does not require conversion and may be printed directly in the original long form. Although it is unlikely that a student running this program will be 32 years old, students no doubt will enter fictitious ages, perhaps ages of parents, relatives, and teachers, thereby experimenting with larger values. This is where the subroutine goes to work!

If you use the Scientific Notation Conversion routine in an educational program, consider inserting commas in numerals, as discussed in Section 7.1.

On the secondary level you may use the Scientific Notation Conversion routine to teach the concept of scientific notation. Try entering a number like 9.4E+12 in the original program. Depending on the need, you may want to alter this code to include negative numbers or very small positive numbers. For further information regarding scientific notation and the rounding procedures built into your computer, refer to the reference manual that accompanies your computer.

ON LINE 7.2

1. List four topics that involve large numbers.
2. Use the Scientific Notation Conversion routine in a program that converts light years to miles. One light year is approximately 5,878,000,000,000 miles.
3. Insert the Commas in Numerals routine (Section 7.1) into the Light Years program created in question 2.

Dollars and Cents 7.3

Educational programs involving the use of money (e.g., making change, monetary values, sales tax) require special treatment when displaying the output. If one multiplies $22.00 by a typical 5 percent sales tax, the basic calculation is 22 multiplied by .05. The result of this calculation, 1.1, has little resemblance to the $1.10 that the student is expecting. Two programming concerns arise when working with printouts involving dollars and cents. (1) The dollar sign should be placed in the answer. (2) Precisely two decimal places are to be reserved for the cents value. The following program lines will insert a dollar sign and print all cents values to the nearest hundredth for any positive number entered.

```
5   INPUT "ENTER A POSITIVE NUMBER ";N
10   GOSUB 5000
15   PRINT N$
20   END
5000 N =   INT (N * 100 + .5)
5010 N$ =   STR$ (N)
5020 L =   LEN (N$)
5030   IF L = 1 THEN N$ = "$.0" + N$
5040   IF L = 2 THEN N$ = "$." + N$
```

```
5050  IF L > 2 THEN N$ = "$" +  LEFT$ (N$,L - 2) + "." +  RIGHT$ (N$,2)
5060  RETURN
```

Enter this routine into your computer and input a positive number when prompted by Line 5. For instance, enter 16.271 and observe $16.27 printed on the screen. Then enter different values as you rerun the program; use values such as 0.04, 5.3, 82.766, 0, and 9.008. A line by line explanation of the subroutine, which begins at Line 5000, clarifies the programming techniques employed.

5000 Multiplies the value assigned to N by 100 and adds .5. Truncates the result to an integer and assigns the final result to the variable N.

5010 Converts the resulting number to a string.

5020 Assigns the length of the newly formed string to the variable L.

5030 If the length of the string is 1, then "$.0" concatenated to the left end of N$ to form a new string assigned to N$.

5040 If the length of the string is 2, then "$." is concatenated to the left end of N$ to form a new string assigned to N$.

5050 If the length of N$ is greater than 2, then the decimal point is placed in the appropriate position, allowing all digits except the last two (L − 2) to be positioned to the left of the decimal while the last two digits will be set to the right of the decimal point. This is done by concatenating four strings together to form N$.

5060 Returns to print the newly created string.

This procedure is a versatile routine that can be used in programs ranging from the most elementary educational program consisting of the summation of money to the most complex business package involving annuities, amortization, and depreciation.

The following brief utility program calculates the sales tax and the total cost of an article. It illustrates the ease with which the Dollars and Cents Conversion routine can be incorporated into a program.

```
100   INPUT "ENTER PURCHASE PRICE ";P
110   INPUT "ENTER SALES TAX RATE ";ST
120 T = P + P * ST
130   GOSUB 5000
140   PRINT : PRINT "THE TOTAL COST IS ";T$
150   VTAB 24: PRINT "ANY MORE CALCULATIONS (Y/N) ";
```

```
160  GET Y$
170  IF Y$ = "Y" THEN  HOME : GOTO 100
180  END
5000 T =  INT (T * 100 + .5)
5010 T$ =  STR$ (T)
5020 L =  LEN (T$)
5030  IF L = 1 THEN T$ = "$.0" + T$
5040  IF L = 2 THEN T$ = "$." + T$
5050  IF L > 2 THEN T$ = "$" +  LEFT$ (T$,L - 2) + "." +  RIGHT$ (T$,2)
5060  RETURN
```

For simplicity, input checks are not included in the program. A finished product should include input checks, as described in Section 4.5, that will guarantee acceptable entries. Here is a line by line explanation.

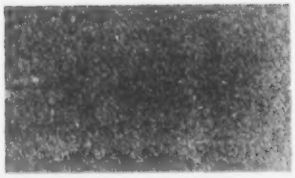
100	Asks the user to enter purchase price and assigns that entry to the variable P.
110	Asks the user to enter the sales tax rate and assigns that entry to the variable ST.
120	Calculates the total cost by adding the purchase price to the sales tax.
130	Goes to the subroutine beginning at Line 5000. Again, this subroutine converts the calculated answer to a proper format including a dollar sign, a decimal point, and two places for the cents value.
140	Prints the total cost as calculated in Line 120 and formatted in the dollars and cents conversion subroutine.
150–180	Quit option routine.
5000–5060	Dollars and Cents conversion subroutine as described earlier in this lesson.

No matter where your programming interests lie, when printing monetary values consider including this routine in your program. If your program requires precision to a place value other than hundredths, minor adjustments to this coding can be made to accommodate various levels of accuracy.

---

1 The Dollars and Cents routine focuses on two formatting requirements. What are they?

ON LINE 7.3



2 Change the Dollars and Cents routine to round to the nearest dime.

3 Write a program that makes change for a $10 bill. Allow the user to enter any positive number less than 10. The resulting output should be the change received from $10.

7.4 Right Justification

Right justification is the formatting process in which strings are printed so that the right character in each string is aligned evenly at the right margin. Right justification can be used to align a column of numbers. For example, if one wants to list the numbers 17.58, 5.92, 103.60, .82, and 37.04 vertically, then right justification is generally required, resulting in the display

```
 17.58
  5.92
103.60
  0.82
 37.04
```

The following program will produce this output. It is based on the use of a CALL −1008 in Line 80. For a discussion of the use of CALLs in formatting, see Section 3.3.

```
10  FOR I = 1 TO 5
20  READ C$(I)
30  NEXT I
35  DATA    17.58,5.92,103.60,.82,37.04
40  FOR I = 1 TO 5
50  HTAB 22
60  FOR J =  LEN (C$(I)) TO 1 STEP  - 1
70  PRINT  MID$ (C$(I),J,1);
80  CALL  - 1008: CALL  - 1008
90  NEXT J
100  PRINT
110  NEXT I
120  END
```

Line 35 contains the five numbers to be read and stored in array C$(I). Line 40 starts the FOR/NEXT loop that prints the numbers. Line 50 tabs to the twenty-second column, and Line 60 begins the loop that prints each digit or character of the first number. This loop

(Lines 60–90) prints each character of the first array element (17.58). Rather than printing the first character, however, Line 70 selects and prints the last character first. Once printed, the cursor automatically moves one column to the right. The cursor is positioned to the left of the last digit printed by using two CALL −1008 statements in Line 80. Line 90 returns control to Line 60, where J is decremented by 1, enabling Line 70 to print the digit preceding the one previously printed. Once the inner loop (Lines 60–90) has completed the cycle determined by the length of the string, the cursor is positioned on the next row (Line 100), and the following array element is then printed in the same manner.

Here is an example of right justification that allows the student user of a math program to enter the sum of two numbers by starting at the units column and then working backward to the tens column and then the hundreds column if necessary.

```
10 X =  INT ( RND (1) * 90 + 10)
20 Y =  INT ( RND (1) * 90 + 10)
30  HTAB 5: PRINT X
40  HTAB 4: PRINT "+";Y
50  HTAB 6
60  GET A$
70  PRINT A$;
80  CALL  - 1008: CALL  - 1008
90  IF A$ <  >  CHR$ (13) THEN B$ = A$ + B$: GOTO 60
100  PRINT
110  IF  VAL (B$) = X + Y THEN  PRINT "CORRECT": GOTO 130
120  PRINT "SORRY, THE CORRECT ANSWER IS ";X + Y
130 B$ = ""
140  PRINT : GOTO 10
```

When this program is run, two random numbers ranging from 10 to 99 will be selected and displayed on the screen. For example, suppose the numbers 72 and 65 are randomly selected. They will be printed on the screen as

```
 72
+65
 ■
```

A blinking cursor prompts the student to enter the units digit, the rightmost digit of the answer. Once a digit is entered, the cursor prompt moves one column to the left, allowing the student to enter the tens digit of the sum:

```
 72
+65
■7
```

Once the tens digit is entered, the cursor is moved to the left of this latest entry:

```
 72
+65
■37
```

When the hundreds digit is typed, the cursor will once again be displayed to the left of the entry, whereby the Return Key can be pressed.

```
 72
+65
■137
```

An explanation of the program that creates the foregoing numeric output follows.

Once the random numbers to be added are selected and positioned on the screen, the cursor is placed at the sixth column (Line 50), and the program awaits the student's response (Line 60). As soon as a key is pressed, the entry is displayed on the screen (Line 70). The semicolon in Line 70 places the cursor to the right of the entry. Since it is necessary to position the cursor to the left of the previous entry, the cursor must be moved back two positions; thus two CALL −1008 statements are needed, and they are listed in Line 80. Line 90 checks to see if the Return key, CHR$(13), has been pressed. If Return has not been pressed, the entry is concatenated with all previous entries and control returns to Line 60, where another digit may be entered. If Return has been pressed, program execution falls to Line 110, where the completed entry is checked and the appropriate response is displayed. Line 130 clears the concatenated entry, and Line 140 returns the student to the beginning of the program.

Right justification is a key formatting process in educational programs involving numeric output. Incorporating the ideas mentioned in this section into your educational computer program will add a polished appearance to the computer screen that allows the user to accurately enter and view numeric data.

ON LINE 7.4

1 List three topics in which Right Justification can be used to enhance the numeric output of a program.

2 Explain the role of a CALL −1008 in the Right Justification routine.

3 Write a program that incorporates the Right Justification routine along with READ and DATA to print the following information.

THE PLANETS AND THEIR AVERAGE DISTANCE FROM THE SUN IN MILLIONS OF MILES

■ MERCURY	36
■ VENUS	67
■ EARTH	93
■ MARS	142
■ JUPITER	484
■ SATURN	887
■ URANUS	1782
■ NEPTUNE	2792
■ PLUTO	3664

Decimal Point Alignment 7.5

Screen display of numbers that are aligned vertically with respect to the decimal point can increase the instructional value of an educational computer program. A wide variety of programs ranging from elementary addition exercises to complicated statistical compilations can benefit by the inclusion of a routine that aligns decimal points in the numeric output of such programs. When numeric output is presented in a clear, precise format, more time can be allotted to the specific task at hand, whether it be adding two numbers, analyzing statistical data, or any other assignments involving numeric data.

If you wish to present a vertical addition exercise consisting of the numbers 6.25 and 13.8, you would want the numbers to be aligned so that the decimal points lie in the same column:

```
 6.25
13.8
```

One method of programming this screen format is to use the TAB function. A run of the following program lines results in the preceding output, in which the decimal points in the two numbers are aligned.

```
10  PRINT  TAB( 8);"6.25"
20  PRINT  TAB( 7);"13.8"
```

Programming problems generally require a solution that can be adapted to a wide range of situations. For this reason the foregoing method is inadequate because the TABs are set to accommodate specific situations: namely, a number having one digit to the left of the decimal point, and a number having two digits to the left of the decimal point. The two-line routine does, however, provide the framework from which a decimal point alignment routine can be created. The general idea is to print the decimal point in a fixed position on the screen and tab back as many places as dictated by the integer portion of the number. If a program line that tabs back a specific number of places can be coded, then a decimal point alignment routine can be created. The program line that generates this positioning should be like this.

```
50  PRINT  TAB( 10 - L);N$
```

L represents the number of digits to the left of the decimal point. N$ is the number to be printed, and the 10 designates the tenth screen column, the desired location of the decimal point; therefore, the only calculation remaining is to count the number of digits to the left of the decimal point.

The following program employs a FOR/NEXT loop to determine L and uses this information to align properly the given data with respect to the decimal point.

```
10  FOR I = 1 TO 8
20  READ N$
30  FOR J = 1 TO  LEN (N$)
40  IF  MID$ (N$,J,1) <  > "." THEN  NEXT J
50  PRINT  TAB( 10 - J);N$
60  NEXT I
70  DATA  5.0,45,-.781,85.6,-12.41,0,-1.8,0.62
80  END
```

The outer FOR/NEXT loop (Lines 10–60) allows for eight numbers in Line 70 to be read. Line 20 reads a number in the DATA statement (Line 70), and the inner FOR/NEXT loop (Lines 30–40) searches for a decimal point in this number. The MID$ function in Line 40 is used to examine as many characters in the string as needed to find the decimal point. Once a decimal point is located, the program exits the loop and goes to Line 50, where TAB is set to 10 minus L, the number of characters in the string up to and including the

decimal point. After the number is printed in the proper position, another iteration of the loop occurs, repeating the same process with the next number in the DATA statement.

What occurs if no decimal point is found in the number? The nature of an Applesoft BASIC FOR/NEXT loop is such that the index variable is also incremented when the loop is completed. For example, after the following Lines 5 and 6 are executed through nine iterations of the loop, the number 10 will be printed.

```
5  FOR I = 1 TO 9
6  NEXT I
7  PRINT I
```

Line 5 initializes I to 1, and Line 6 increments I by 1 before returning execution to Line 5. When I is 9, Line 6 increments I to 10 before exiting the loop. Line 7 then causes the number 10 to be printed on the screen. With this assignment, I = 10, in mind, we can re-examine the Decimal Point Alignment routine.

If the number assigned to N$ in the Decimal Point Alignment program lacks a decimal point, the loop in Lines 30 and 40 will be completed before L is determined. For example, if the number (N$) is 45, the loop formed by Lines 30 and 40 will iterate twice, since the string is two characters long and does not contain a decimal point. Upon completion of the loop, L will be set to 3 and TAB will be moved back three places. Since the calculation of L includes the position of the decimal point, this is the same number of places that a number will be tabbed back if it contained two characters to the left of its decimal point. Thus an idiosyncrasy in the workings of a FOR/NEXT loop allows for correct output without use of further conditional statements.

The Decimal Point Alignment routine uses READ and DATA statements to assign values to the variable N$. This allows for convenient illustration and explanation of the program. However, your use of this routine in an educational computer program may very well demand a different method of assigning numbers to a string variable.

The following program uses the INPUT statement to assign as many as 100 numbers to the array, N$(C), before a printout of these numbers is displayed on the screen.

```
10  DIM N$(100)
20 C = 1
30  INPUT "ENTER A NUMBER ('S' TO STOP) ";N$(C)
40  IF N$(C) <  > "S" THEN C = C + 1: GOTO 30
50  HOME
```

```
60  FOR I = 1 TO C - 1
70  FOR J = 1 TO  LEN (N$(I))
80  IF  MID$ (N$(I),J,1) <  > "." THEN  NEXT J
90  PRINT  TAB( 20 - J);N$(I)
100  NEXT I
110  END
```

This program uses an array, N$(C), to enable the student to enter numbers during the program run. Line 10 dimensions the array, allowing up to 100 numbers to be entered. Line 20 sets the counter to 1. The user is then asked to enter a number, and the entry is assigned to the first variable in the string array, N$(1). Line 40 checks to determine if more entries are to be made. Included in Line 30 is the opportunity for the user to stop the entry process. If S, for stop, is typed, then the condition in Line 40 will not be met, and execution will continue to Line 50. However, if a number is entered, the counter in Line 40 is incremented by 1 and program execution returns to Line 30, where the next entry can be input. Once the user types S, the input loop terminates and the screen is cleared (Line 50). Line 60 initiates the loop that prints all the entries in an aligned format. Although there are C entries, only C −1 of these will be printed, because the last entry (the one that stops the input loop) is S. The Decimal Point Alignment routine is coded in Lines 70–90. Except for the use of an array and a repositioning of the TAB to Column 20, it is identical to the alignment routine explained earlier.

---

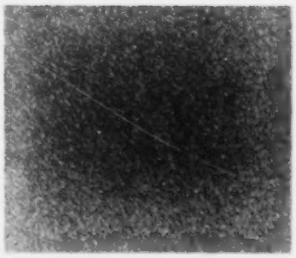
1 Identify three topics that can utilize the Decimal Point Alignment routine in an educational computer program.

2 Use the Decimal Point Alignment routine to produce the following screen display concerning nutrition information (Nutri Data) for a certain breakfast cereal.

GRAMS PER SERVING:

PROTEIN	6
CARBOHYDRATE	31.6
FAT	0
CHOLESTEROL	.015
SODIUM	.34
POTASSIUM	1.02

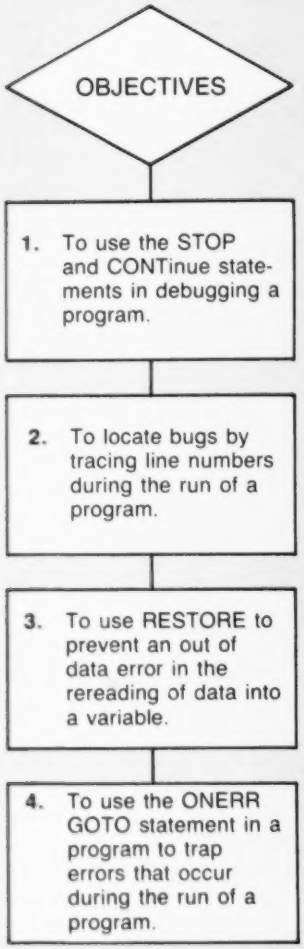
3 Write a program that will accept as input the name of a mineral, its hardness, and its specific gravity. The output should look similar to the following.



MINERAL	HARDNESS	SPECIFIC GRAVITY
GOLD	2.5	14.5
DIAMOND	10	3.5
SILVER	3	11
OPAL	5.98	1.93

CHAPTER EIGHT

DEBUGGING


OBJECTIVES
1. To use the STOP and CONTinue statements in debugging a program.
2. To locate bugs by tracing line numbers during the run of a program.
3. To use RESTORE to prevent an out of data error in the rereading of data into a variable.
4. To use the ONERR GOTO statement in a program to trap errors that occur during the run of a program.

Developing your ability to locate and remove programming errors will reduce the time involved in completing the final form of an educational computer program and will reduce the sweat and toil that often goes into the debugging process. This chapter explains five statements that aid this process: STOP, CONTinue, TRACE, RESTORE, and ONERR GOTO. Through effective use of STOP and CONTinue, you will be able to locate and eliminate errors such as the incorrect assignment of a value to a variable. You will also learn how to use the TRACE command to locate the number of a line containing an error that appears as a program is running.

The correct programming of READ/DATA and the use of RESTORE to prevent errors in rereading data is explained. Section 8.4 shows you how to use the ONERR GOTO statement that traps an error during the run of a program and sends the program to a line coded to rectify the error.

8.1 STOP and CONTinue

You already know that an END statement in BASIC causes the program to cease execution and return control of the computer to the user. If your Applesoft BASIC program does not contain an END statement, it will end automatically when it runs out of statements. The STOP statement can be used in debugging a program to terminate execution of the program. When this statement is used, a message indicating the line number at which execution ceased is printed before control is returned to the user. The user can then use the immediate execution mode to check the values of different variables and arithmetic expressions to determine if any are incorrect. To resume execution of the program at the statement immediately following STOP, the user simply types CONT, the BASIC command for continue, and presses Return.

Let's give the Apple a simple math problem that we will investigate by later inserting a STOP statement at a particular point in the program:

```
10  LET X = 5
20  PRINT  - X * 2
30  PRINT "THAT'S APPLE LOGIC"
40  END
```

Many people are quite surprised to find that, on typing RUN, the Apple returns an answer of 25. If you run the same program on some

other computers, the answer returned is −25. Obviously, the Apple is using a different order of arithmetic operations than the other computers. Inserting a STOP statement at Line 22 will allow us to use immediate mode and print the content of variable X and the result of the computation, −X ^ 2.

```
10  LET X = 5
20  PRINT  - X ^ 2
22  STOP
30  PRINT "THAT'S APPLE LOGIC"
40  END
```

After you type RUN, the program will terminate at Line 22, and the message BREAK IN 22 will be printed. At this point type PRINT X and press Return. The result, 5, will be printed showing that the variable X does indeed hold the expected value. Now type, PRINT −X, and the result, −5, will be printed, showing that the Apple has attached a negative sign to the 5. When a minus sign is used to indicate a negative number, it is called a unary minus sign to distinguish it from the minus sign that means subtract, as in 7 − 3.

Now type PRINT −X ^ 2, and 25 will appear. The Apple first attached the negative sign to the 5 before squaring the number. Other computers would square the number first and then attach the negative sign. Now, we type CONT and press Return. The program resumes execution at Line 30, at which point THAT'S APPLE LOGIC is printed. Then the program terminates execution at Line 40.

We have seen that the order of precedence of arithmetic operations for the Apple is unary minus first, exponentiation second. The precedence for these two operations on some computers is just the opposite: exponentiation is performed first, followed by unary minus. What is the moral of all this? Remember to use parentheses whenever you are not certain of the order of arithmetic, relational, or logical operations. For example, to force the Apple to perform the exponentiation before the answer is assigned a negative value with the unary minus sign in the previous program, we would write Line 20 as

```
20  PRINT  - (X ^ 2)
```

A STOP statement can be placed at any location in a program. Placing STOPs at strategic points will help you debug (find and correct program errors) in your educational program. When all the bugs have been removed, the statements containing STOP are deleted, and the program is ready to be used.

Two types of errors can appear in your programs: syntax errors and logic errors. A syntax error occurs when you, for example, misspell a command in BASIC, such as typing PRUNT for PRINT, or when you incorrectly construct a statement, such as typing X+Y=Z rather than Z=X+Y (only one variable may appear to the left of the assignment symbol). Syntax errors are usually easily identified because the BASIC compiler or interpreter will generally detect them and print an error message, ?SYNTAX ERROR IN Y, where Y is the line number of the statement in which the error occurred. Syntax errors will not be detected until you type RUN and the computer attempts to execute the incorrect statement.

You will also discover that your computer will print specific, cryptic messages for various types of errors. For example: ?NEXT WITHOUT FOR ERROR, ?OUT OF DATA ERROR, ?RETURN WITHOUT GOSUB ERROR, and ?TYPE MISMATCH ERROR. Most of these can be corrected by careful inspection of the program listing. Consult an Applesoft programming manual for a complete list of error messages.

Logic errors are quite another matter. They are often very difficult to locate because they are not detected by the compiler or interpreter. An example of a logic error occurs if you type Y=A/B when you intended to type Y=B/A, and the error went unnoticed. We cannot expect the computer to realize what you intended! Then how does one find these sometimes very elusive logic errors? Use STOP, as we demonstrated earlier in the program containing PRINT −X ^ 2.

Here is another example of the use of STOP. The following simple program was written to compute and print the result of the addition of two integers divided by a third integer: for example, add 25 to 10 and divide the result by 5.

```
10  PRINT "PLEASE ENTER THE FIRST INTEGER."
20  INPUT NUM1
30  PRINT "PLEASE ENTER THE SECOND INTEGER."
40  INPUT NUM2
50  PRINT "PLEASE ENTER THE THIRD INTEGER."
55  INPUT NUM3
60 R = NUM1 + NUM2 / NUM3
70  PRINT "THE SUM OF TWO INTEGERS DIVIDED BY A THIRD INTEGER IS ";R
80  END
```

If we run this program and enter 25 for NUM1, 10 for NUM2, and 5 for NUM3, the computer will inform us that the answer, R, is 6. Clearly the answer is 7. What's up? Let's modify the program by

inserting STOP statements at Lines 22, 42, 58, and 65. The revised program is shown here.

```
10  PRINT "PLEASE ENTER THE FIRST INTEGER."
20  INPUT NUM1
22  STOP
30  PRINT "PLEASE ENTER THE SECOND INTEGER."
40  INPUT NUM2
42  STOP
50  PRINT "PLEASE ENTER THE THIRD INTEGER."
55  INPUT NUM3
58  STOP
60 R = NUM1 + NUM2 / NUM3
65  STOP
70  PRINT "THE SUM OF TWO INTEGERS DIVIDED BY A THIRD INTEGER IS ";R
80  END
```

Now we rerun the program. After we enter 25 for NUM1, the program ceases executing at Line 22, and the message BREAK IN 22 is printed. At this point, control of the computer is returned to the user, and we type PRINT NUM1, press Return, and the computer prints 25. Fine, no problem here. Now we type CONT, press Return, and the program begins executing at Line 30. We enter 10 for NUM2. Again the program ceases executing, this time at Line 42. We enter PRINT NUM2, and the computer prints 10. Still no problem. Again we enter CONT, and program execution begins at Line 50. When the program stops executing at Line 58, we enter PRINT NUM3, and the computer prints 5. Still no problem. We enter CONT, and the program comes to a halt at Line 65.

At this point we will check the values of those variables again just to make certain nothing strange is occurring. We enter PRINT NUM1, NUM2, NUM3, and the computer prints a 5 as the value of each of the three variables. This cannot be, we say. Yes it can, because Applesoft BASIC recognizes only the first two characters in a variable name. Actually, we do not have three different variables. We have only one variable, NU. In the first INPUT statement we entered 25, and this value was stored in NU. When we came to the second INPUT statement, we entered 10, and this value replaced the 25 stored in NU. On the third INPUT we entered 5, and this value replaced the 10 stored in NU. So the calculation in Line 60 is actually R = NU + NU/NU, and NU = 5. Therefore, our result should be the sum of 5 + 5, divided by 5, which is 2. But the computer prints 6. We must have another problem with logic.

In Lines 20, 40, and 55 we will change the variable names to N1, N2, and N3, respectively. Now we are certain there are no problems

with variable names, so we remove the STOP statements in Lines 22, 42, and 58. We leave the STOP statement in Line 65. Line 60 is changed to R = N1 + N2/N3.

Again we run the program, and it ceases execution at Line 65. At this point we type PRINT N1, N2, N3 and get 25, 10, and 5, respectively. No problem with the values of the variables now. The problem must then be with the arithmetic statement we constructed at Line 60. Let's examine it more closely. Now we see it. We forgot that BASIC follows a specified order of operations when doing arithmetic computations. Specifically, division is performed before addition. In our statement, the computer first divides N2 by N3 and then adds N1. As far as the computer is concerned, the answer 6 is correct because 5 + 5/5 is 5 + 1, which is 6. We can force the computer to do the addition first by placing parentheses around N1 and N2:

```
60 R = (N1 + N2) / N3
```

Admittedly, this was a rather simple example to use to explain the use of multiple STOP statements to debug programs. But the procedure is exactly the same when you are attempting to locate errors in programs containing hundreds of lines.

---

1 Use the STOP feature to debug the following program, which is intended to compute the average of four numbers entered by the user.

```
15 LET S = 0
20 FOR I = 1 TO 4
30 INPUT N
40 LET S = S + N
50 NEXT I
60 LET A = S / I
70 PRINT "THE AVERAGE OF YOUR FOUR NUMBERS IS ";A
80 END
```

2 Use the STOP feature to investigate the output of this program.

```
10  LET X = 5
20  PRINT  - X ^ 2 - X ^ 2
30  END
```

3 Distinguish between STOP and END statements in a BASIC computer program.

4 How does one decide where to place STOP statements in an educational computer program?

Tracing Line Numbers 8.2

Section 8.1 explained the use of STOP coupled with CONT to debug programs. The TRACE command is also quite useful in debugging programs, especially those containing loops, subroutines, and conditional statements such as IF-THEN. TRACE causes the line number of each statement to be displayed on the screen as the statement is executed, thereby allowing you to see if the program is performing the desired sequence of operations. To use the TRACE feature, merely type the word TRACE and press Return in immediate execution mode before running the program, which has previously been entered into the computer. To turn the TRACE feature off, simply type NOTRACE and press Return.

Here is a simple BASIC program to illustrate the use of TRACE.

```
10  PRINT "ONE"
20  PRINT "TWO"
30  PRINT "THREE"
40  END
```

Without using TRACE the output of this program is

```
]RUN
ONE
TWO
THREE
```

After typing TRACE, the output is

```
]RUN
#10 ONE
#20 TWO
#30 THREE
#40
```

Both the line numbers and output are printed on the screen. The #10 shows that the statement in Line 10 is executed first, the #20

indicates that the statement in Line 20 is executed next, and the #30 indicates that the statement in Line 30 is executed after Line #20. The #40 indicates that the final statement to be executed is END, which terminates the program.

What happens if a program line contains multiple statements, such as Line 10?

```
10  PRINT "ONE": PRINT "TWO": PRINT "THREE": END
```

Let's use the TRACE feature to follow the execution of this line.

```
]RUN
#10 ONE
#10 TWO
#10 THREE
#10
```

The TRACE command traces the execution of each statement in a line. Repeated printing of #10 indicates that the statements in Line 10 are executed in sequential order, as would be expected. Since there are four separate statements in Line 10, the number 10 is printed four times. Remember, TRACE traces the flow of program execution by displaying the line number of each statement as it is executed.

Here is another simple program that contains a conditional statement. We will use the TRACE feature to locate a programming error.

```
5  PRINT "GUESS MY LUCKY NUMBER."
10  PRINT "PLEASE TYPE AN INTEGER FROM 1 TO 10."
20  INPUT A
30  IF A = 6 THEN  PRINT "YOU GUESSED MY LUCKY NUMBER": PRINT "GOOD"
35  PRINT "YOU MISSED IT"
40  END
```

Using TRACE, we get the following output if a 3 is entered.

```
]RUN
#5 GUESS MY LUCKY NUMBER.
#10 PLEASE TYPE AN INTEGER FROM 1 TO 10.
#20 ?3
#30 #35 YOU MISSED IT
#40
```

No bug here. Notice that #30 indicates that the IF part of the IF-THEN statement is executed but not the THEN part, because the

IF condition (IF A = 6) is not satisfied by an input of 3 (A = 3). Also we see that the second statement in Line 30, PRINT "GOOD", is not executed.

On a second run of the program a 6 is entered at the ?.

```
]RUN
#5 GUESS MY LUCKY NUMBER.
#10 PLEASE TYPE AN INTEGER FROM 1 TO 10.
#20 ?6
#30 YOU GUESSED MY LUCKY NUMBER
#30 GOOD
#35 YOU MISSED IT
#40
```

There's the bug! The answer 6 cannot be a correct guess and at the same time be incorrect. The #30 indicates that the IF-THEN condition is met. Both PRINT statements in Line 30 are executed. The #35 indicates that the next statement executed is at Line 35, which is certainly not what we have in mind. For the routine to run correctly, we must change Line 30 by adding a GOTO statement:

```
30 IF A = 6 THEN PRINT "YOU GUESSED MY LUCKY NUMBER.":PRINT "GOOD": GOTO
   40
```

Type in the following program, which has repeated calls to the subroutine at Line 1000, and run it.

```
10   GOSUB 1000
20   PRINT "ONE"
25   GOSUB 1000
30   PRINT "TWO"
40   GOSUB 1000
50   STOP
1000   PRINT "OK "
1010   RETURN
```

The output is

```
]RUN
OK
ONE
OK
TWO
OK

BREAK IN 50
```

Now use the TRACE feature and let the computer help in following the sequence of execution. Type TRACE and press Return. Now run the program. The output is

```
]RUN
#10 #1000 OK
#1010 #20 ONE
#25 #1000 OK
#1010 #30 TWO
#40 #1000 OK
#1010 #50
BREAK IN 50
```

Now type NOTRACE and press Return to turn the TRACE feature off. Inspection of the output shows that the statement at Line 10 is executed first, followed by the statement at Line 1000, at which point OK is printed. Next, the statement at Line 1010 is executed, followed by the statement at Line 20, at which point ONE is printed. Then the statement at Line 25 is executed, followed by the statement at Line 1000, at which point OK is printed. The output clearly indicates the exact sequence of execution of the statements. Note that Line 1000 shows up three times; this is correct because there are three calls to this subroutine. Certainly the simple program is not too difficult to follow even without help from the TRACE feature. But if your programs contain more complicated branching structures or subroutines, this is the time when TRACE can be indispensable in locating elusive bugs.

Here is another program involving subroutines. A run of the program will show that it contains bugs, which you can locate with TRACE.

```
10   GOSUB 100
15   GOTO 500
20   PRINT "AND WE WILL.";
100   PRINT "WE MUST ";
110   GOSUB 200
120   RETURN
200   PRINT "FIND THE ";
210   GOSUB 300
220   RETURN
300   PRINT "BUGS IN ";
310   GOSUB 400
320   RETURN
400   PRINT "THIS PROGRAM,";
410   RETURN
500   END
```

This program illustrates the feature of one subroutine calling another. The intended output is WE MUST FIND THE BUGS IN THIS PROGRAM, AND WE WILL. But the actual output is WE MUST FIND THE BUGS IN THIS PROGRAM,. Again we use the TRACE feature to study the execution of the statements.

```
]RUN
#10 #100 WE MUST #110 #200 FIND THE #210 #300 BUGS IN #310 #400 THIS PROGRAM,#4
0 #320 #220 #120 #15 #500
```

Inspection of the output shows that the program branches as follows: Line 10 to Line 100 and prints WE MUST; Line 110 to Line 200 and prints FIND THE; Line 210 to Line 300 and prints BUGS IN; and Line 310 to Line 400 and prints THIS PROGRAM,. So far so good! But where is the elusive AND WE WILL? Notice that the line number, 20, for this statement does not show in the output, meaning that it was never executed. Further inspection shows that the program branched as follows: Line 410 to 320 to 220 to 120 to 15 to 500. Now the reversal of the PRINT and GOTO statements reveals itself: Lines 15 and 20 have inadvertently been interchanged in the listing. These lines should be

```
15  PRINT "AND WE WILL.";
20  GOTO 500
```

Here is a trace of the corrected program:

```
]RUN
#10 #100 WE MUST #110 #200 FIND THE #210 #300 BUGS IN #310 #400 THIS PROGRAM,#4
0 #320 #220 #120 #15 AND WE WILL.#20 #500
```

The sample programs in this section demonstrate the usefulness of the TRACE feature in debugging a program. Using the STOP statement together with TRACE provides a means of quickly locating and correcting programming errors.

---

1 How can the TRACE feature be used together with the STOP feature to debug an educational computer program?

2 When using TRACE it is important to distinguish between a line and a statement in a BASIC program. Explain.

3 Use TRACE to investigate this program:

```
10  PRINT "IT'S"
20  PRINT "TIME"
30  PRINT "TO GO"
40  HOME
50  PRINT "GOODBYE"
```

4 Which types of programming errors are more easily found using the STOP feature? Which types are more easily found using the TRACE feature?

5 Use the TRACE feature to follow the flow of this program, which is intended to determine the relative position of two letters in the alphabet. Locate the error.

```
5  FOR I = 1 TO 5
10  INPUT FL$,SL$
20  GOSUB 500: GOTO 50
30  GOTO 10
35  NEXT I
50  END
500  IF FL$ < SL$ THEN  PRINT FL$;" COMES BEFORE ";SL$: GOTO 800
600  IF SL$ < FL$ THEN  PRINT SL$;" COMES BEFORE ";FL$: GOTO 800
700  PRINT FL$;" AND ";SL$" ARE THE SAME LETTER."
800  RETURN
```

6 Compare and contrast the use of STOP and TRACE statements to debug programs.

7 Would STOP or TRACE be more useful in finding the error in this program?

```
10  PRINT "ENTER TWO POSITIVE INTEGERS"
20  INPUT N1,N2
30  LET A = N1 + N2 / 2
40  PRINT "THE AVERAGE OF THE TWO NUMBERS IS ";A
```

8.3 READ/DATA and RESTORE

Two statements, READ and DATA, working in tandem to assign values to variables, can be at the source of several intriguing

programming errors. Since READ and DATA do work together, many BASIC students believe these two statements should either be listed sequentially, or, with FOR/NEXT loops, the DATA statement should be listed immediately following the NEXT statement. The following two listings illustrate this placement of the DATA statement in relation to the READ statement.

```
10  READ N1$,N2$
20  DATA RUTH,BABE
```

```
10  FOR I = 1 TO 2
20  READ N$(I)
30  NEXT I
40  DATA RUTH,BABE
```

Although not required, the technique of placing the DATA statement in close proximity to the corresponding READ statement has its advantages. For instance, the programmer can readily locate the strings or numerical values assigned to specific variables, making it easy to locate and change data when necessary. However, coding READ and DATA statements in this fashion can create several programming errors for which the computer does not print an error message. For example, because of an unconditional branch in Line 40, the following program will not print ONE, THREE, the expected output when the program is run.

```
10  READ A$
20  DATA ONE
30  PRINT A$
40  GOTO 80
50  READ A$
60  DATA TWO
70  PRINT A$
80  READ A$
90  DATA THREE
100  PRINT A$
110  END
```

Figure 8.1
Routine from The Number Machine program that reads nine words (numbers) into the subscripted variable U$(I).

When running the foregoing program, you may expect the output to be ONE, THREE because of the routine locations of the DATA statements. This expectation is based on the mistaken belief that the data read is always located nearest the READ statement being executed. Thus, when Line 80 is executed, one may incorrectly associate the data element in Line 90 with the READ statement in Line 80.

However, data elements are read sequentially in a program, regardless of the location of the READ statements within the program. The reason is that in BASIC there is an invisible pointer that points to the data item in a DATA statement to be read by an associated READ statement. As READ statements are executed, the pointer moves from one data item to the next until it reaches the last item. Thus the first value read will be ONE, the initial element in the first DATA statement (Line 20). The second READ statement, executed by Line 80, will then read TWO, the next data element in the program (Line 60). Consequently the output of the program is ONE, TWO, not the expected output of ONE, THREE.

When you write a long program involving many READ and DATA statements, a mistake of this nature can be a most irritating problem to debug. How then can such a mistake be avoided? One method is to list all data referencing the same variable in a single DATA statement:

```
10  READ A$
20  PRINT A$
30  GOTO 60
40  READ A$
50  PRINT A$
60  READ A$
70  PRINT A$
80  DATA ONE,TWO,THREE
90  END
```

Recognizing that data are always read sequentially—that is, ONE will be read first, TWO second, and THREE last (if at all)—you can now read through the program and readily observe the correct output. Line 10 reads the first data element, ONE, and assigns it to A$. Line 20 prints ONE. Line 30 transfers program execution to Line 60, where the second data element, TWO, is read and assigned to A$. This string, TWO, is printed, and the program ends.

Notice the lack of confusion with Lines 40 and 50! They were totally disregarded, as was THREE, the third data element. Line 30 transfers program execution to Line 60, bypassing Lines 40 and 50. Since only two READ statements are executed, only two data items will be read—the first two elements in the DATA line, ONE and TWO.

Unexpected results can also occur when DATA statements are improperly coded. The following program illustrates a situation in which the first names of an individual are expected to be printed, but the result is something else. Read through the program, jot down the output you expect, then type the program into your computer and run it.

```
10  READ F1$,L1$
20  DATA MARK TWAIN
30  READ F2$,L2$
40  DATA SAMUEL,CLEMENS
50  PRINT F1$,F2$
60  DATA TOM SAWYER
70  END
```

Surprise! Once again, a program is executed in which no error message appears, yet the result is not the anticipated one. Because of the absence of a comma in Line 20, F1$ is assigned the one data string MARK TWAIN, meaning that L1$ is assigned the next data element, SAMUEL, found in Line 40. When Line 30 is executed, the next available data element is CLEMENS, meaning that F2$ becomes CLEMENS and L2$ becomes TOM SAWYER. The output generated by Line 50 will be MARK TWAIN CLEMENS, an output not easily ascertained through a cursory reading of the program.

Omit Line 60 and rerun the program. This time the computer prints the error message, ?OUT OF DATA ERROR IN 30. Did you ever think the appearance of an error message on the screen would be a welcome sight? The reason for this message is the lack of data for L2$ in Line 30. The omission of a data element in Line 60 leaves only three data items to be read by four variables. Since L2$ is the fourth variable to be used in a READ statement, it is now lacking a corresponding data element; thus the out of data error message appears.

READ/DATA statements are also at the core of other perplexing programming errors. Anyone who has ever tried his or her hand at programming has frequently encountered that infamous error message ?SYNTAX ERROR IN X, where X is the actual number of the program line containing the error.

The following program is written for those who find comfort in syntax error messages that designate the number of the line containing the error. Run this program and use the error message printed on your computer screen to locate the bug.

```
10  FOR I = 1 TO 4
20  READ A
```

```
30  PRINT "YOUR LOCKER NUMBER IS ";
40  PRINT A
50  NEXT I
60  DATA 123B,9880,457I,6693
70  END
```

Welcome to the real world of syntax errors. A run of this program greets you with the message ?SYNTAX ERROR IN 60. The obscure reason for this syntax error does not necessarily lie in Line 60 at all. Line 20 is the actual source of the problem. A numerical variable, A, is used to attempt to read a string in Line 60. Converting all data elements in Line 60 to numerical values will solve the problem; but a better way to avoid this problem is to always read values using a string variable. Simply change Line 20 to READ A$ and Line 40 to PRINT A$ and rerun the program. Once again, the idea of avoiding numeric variables has surfaced. As pointed out in Section 4.5, it is a wise practice to use string variables in most instances and then convert to numeric values if necessary.

Sometimes an out of data error occurs if a program attempts to reread data. This may happen when a user is given the option to go back over material presented earlier. Execution of the option branches the program back to a point where data may be reread by one or more READ statements. An illustration of this situation is found in the following listing.

```
10  READ A$
20  DATA HELLO
30  PRINT A$
40  PRINT "RUN AGAIN (Y/N)";
50  GET Y$
60  IF Y$ = "Y" THEN 10
70  END
```

If the student opts to run this program a second time by typing Y in response to the message in Line 40, the program flow transfers to Line 10, where an attempt is made to reread the data in Line 20. Since this data has already been read during the initial run of the program, an attempt will be made to read the next data element. Finding none, an ?OUT OF DATA ERROR IN 10 message will appear, and the program run will be aborted.

This type of programming error can be avoided in several ways. One method is to change Line 60 to IF Y$ = "Y" THEN RUN. Since the RUN statement clears all variables and resets the data pointer to the first element, this statement proves to be a valid solution to the out of data error problem. However, if your educational

computer program contains a counter or a tally, as described in Section 5.2, then these values would be reset to zero by the RUN statement. To avoid the initialization of such variables, you can use a statement that affects only DATA statements. This statement, RESTORE, resets the data pointer to the first data element without clearing any variable values. A RESTORE statement, as illustrated in the following program, should be used in all instances where data needs to be reread but other variables are to remain intact.

```
10 R = 1
20  READ A$
30  DATA HELLO
40  PRINT A$
50  PRINT "RUN AGAIN (Y/N)"
60  GET N$
70  IF N$ = "N" THEN  PRINT "I PRINTED HELLO ";R;" TIMES": END
80 R = R + 1
90  RESTORE
100  GOTO 20
```

Line 10 sets the counter, in this instance representing the number of runs of the program, to 1. Line 20 reads the data element, HELLO, found in Line 30 and assigns it to the variable A$. Line 40 prints HELLO, and Line 50 asks the user if another run is desired. If N is entered, then the program ends, informing the user of the number of completed runs. If the user chooses to run the program again, then program execution drops to Line 80, where the counter is incremented by 1. The RESTORE statement in Line 90 then resets the data pointer to the first data element, in this case the only data element, before Line 100 transfers program execution back to Line 20. At Line 20 the data can again be read, thanks to the RESTORE statement.

Using READ and DATA statements to assign values to variables is a fundamental, tried and true programming technique. Combined with the RESTORE statement, they can provide the foundation for many creative educational programs. Maintaining a standard procedure for incorporating these statements into your programs will help you avoid the errors mentioned in this section. Specifically, try to read all data elements with string variables and list all data elements referencing the same variable in one DATA statement. These suggestions, along with proper use of the RESTORE statement, should free you from any errors associated with the READ and DATA statements.

1 What are the advantages and disadvantages of placing DATA statements in close proximity to READ statements?

2 What occurs when a numeric variable is used to read a string data element?

3 What is the advantage of using RESTORE rather than RUN when rereading data?

4 Find and correct the error in the following program:

```
10  READ F$,M$,L$
20  DATA ALEXANDER,GRAHAM,BELL
30  DATA THOMAS,ALVA,EDISON
40  PRINT L$,F$,M$
50  IF L$ = BELL THEN 10
60  END
```

5 What is the purpose of commas in a DATA statement?

6 Use the RESTORE statement to fill array A$(X) with the same data element five times. Once this is done, use another DATA statement to READ a second variable into A$(6).

8.4 ONERR GOTO

Errors that occur during the execution of a program can be a serious problem because they terminate the program. Examples of potential programming errors include branching to a nonexistent line number, trying to divide by zero, and attempting to read more data than provided for in DATA statements. The Applesoft BASIC statement ONERR GOTO can be used to trap programming errors so as not to interrupt the program. This internal debugging device can be placed anywhere in a program before an anticipated error occurs. It is recommended that the error trap be coded at the beginning of the program, well before its expected need.

A brief example illustrating the placement and use of the ONERR GOTO statement follows:

```
10  ONERR  GOTO 200
20  HOME
30 PRNT"HOW OLD ARE YOU ?"
40  INPUT A
50  END
200  PRINT "PRINT WAS MISSPELLED IN LINE 30"
```

When the program is run and the computer attempts to interpret the unfamiliar instruction in Line 30, PRNT, the normal ?SYNTAX ERROR IN 30 message is replaced by the more definitive message found in Line 200, PRINT WAS MISSPELLED IN LINE 30. Execution of Line 200 is activated by the ONERR GOTO statement in Line 10. Any programming error or DOS error can be trapped by the ONERR GOTO statement. Simply insert a program line such as Line 10 into your program. If an error occurs, this line diverts program execution to the line number referred to in the ONERR GOTO statement.

In the developmental stages of coding an educational program, insertion of the ONERR statement can allow for execution without interruption due to undefined statement errors. As a program is written, certain subroutines that perform needed tasks may be foreseen, but the actual coding may have to await further information.

The following example represents a situation in which a program segment is being developed without inclusion of specific line numbers that are necessary to complete all of the required subroutines.

```
10  ONERR  GOTO 8000
600  FOR A = 1 TO 3
605  PRINT "IDENTIFY THE PREPOSITION"
610  PRINT "IN THE FOLLOWING SENTENCE"
615  PRINT
620 X =  INT ( RND (1) * 5 + 1)
625  ON X GOSUB 1000,2000,3000,4000,5000
630  INPUT AN$
635  IF AN$ = PR$ THEN  PRINT "CORRECT": GOTO 645
640  PRINT "SORRY, THE PREPOSITION IS => ";PR$
645  NEXT A
999  END
1000  PRINT "THE MONEY IS UNDER THE TABLE."
1010 PR$ = "UNDER"
1020  RETURN
2000  PRINT "TABLES ARE MADE OF WOOD."
2010 PR$ = "OF"
2020  RETURN
8000  GOTO 620
```

The intent of this program is to display a sentence containing a preposition and to allow the user to correctly identify it. Though not complete, the program can be run without a break caused by a programming omission. However, the lack of numbered statements such as 3000 or 4000 will trigger an ?UNDEF'D STATEMENT ERROR IN 625 message if the value of X is 3, 4, or 5. Line 10 precludes this interruption. Control goes to Line 8000, a simple GOTO statement that transfers program execution back to Line 620, where another number can be selected and used to branch to the proper subroutine.

Sample runs of partially completed programs can be made without disruption by the standard error messages. These sample runs enable the programmer to view and format the frame being created without completing all the routines involved. Once the program is complete, the ONERR GOTO statement should be deleted, along with any statements used in conjunction with it. In the routine previously illustrated, Lines 10 and 8000 would be deleted as soon as satisfactory completion of the segment is attained, since other problems may arise when the ONERR GOTO statement is used in conjunction with GOSUB and RETURN.

The ONERR GOTO statement can also be used in the development of an educational program that utilizes an equation or formula in which division by zero is likely to occur. The following routine allows the program to continue even though division by zero has been attempted.

```
10  ONERR  GOTO 500
100  FOR I =  - 3 TO 3
110 Y = (I + 1) / (I * 2 - 4)
120  PRINT "X = ";I,"Y = ";Y
130  NEXT I
499  END
500 I = I + 1
510  PRINT
520  RESUME
```

Here is a line-by-line explanation.

10 The ONERR instruction that diverts program flow to Line 500 if any type of programming error is encountered.
100 Assigns and increments the value of I.
110 Defines an equation whose denominator (I * 2 – 4) will be zero when I is either 2 or –2.
120 Prints X as the value of I and the calculated value of Y.
130 Returns to Line 100 for another pass through the loop.

500-520 The ONERR routine that is executed when an error has occurred. In this instance the anticipated error is division by zero, which occurs when I is −2 and 2. In both of these instances Line 500 will increment the value of I by 1, move the cursor down one line and resume program execution at Line 110, the program line in which the error occurred.

Line 520 contains a statement that may be unfamiliar to many programmers: RESUME is an Applesoft statement that is used in conjunction with the ONERR GOTO statement. The sole purpose of the RESUME statement is to transfer program execution back to the beginning of the program line in which the error occurred. Use of RESUME should only take place when the error is both anticipated and corrected. Using RESUME without correcting the error will result in a continuous and futile loop. For example, if Line 500, I = I + 1, was changed to a simple PRINT statement and the previous example were run, the division by zero error would be trapped but the mistake would remain uncorrected. Program execution would return to Line 110 via the RESUME statement, and the error would reoccur, once again sending the program to the ONERR routine beginning at Line 500. This scenario would continue ad infinitum, effectively transforming the ONERR GOTO statement from an error handling routine to an error routine. Use RESUME only when the error is corrected in the routine executed by the ONERR GOTO statement.

As the program is refined and expanded, the occasional encounters of division by zero can be avoided by selecting variable values that generate expressions other than zero. Until the formulas or equations have been selected and tested, however, the ONERR GOTO routine can be used and should be regarded as a valuable aid in the development of programs requiring division by a variable.

Some educational programs are structured to read an abundance of data into memory. As these programs are revised and expanded, the contents as well as the quantity of data might change. Rather than redimensioning arrays or editing the final value in FOR/NEXT loops, one may use the ONERR GOTO statement to count the data entries and make the necessary adjustments to insure a proper run of the program.

The following routine illustrates how the ONERR GOTO statement can be used to count the number of data elements so that an array can be dimensioned and a FOR/NEXT loop can be used to read the data into this array.

```
5  ONERR  GOTO 80
20  READ A$
25  DATA  AL,BO,CAV,DOT,ED,FERN
30 N = N + 1
40  GOTO 20
80  RESTORE
85  POKE 216,0
90  DIM D$(N)
100  FOR X = 1 TO N
110  READ D$(X)
115  PRINT D$(X)
120  NEXT X
```

After Line 5 sets the expected error trap, which is to catch the ?OUT OF DATA IN 20 ERROR, the GOTO loop in Lines 20–40 continually counts the number of entries read. Each time through the loop the N in Line 30 is incremented by 1. Once the program is out of data and the subsequent error is trapped, program execution branches to Line 80, which restores the data pointer to the first data element. Line 85 is a special POKE that cancels the ONERR GOTO statement, clearing the error flag so that normal error messages can occur. Thus if a future programming error should be encountered, execution will not once again return to Line 80. Having calculated the number of entries in the data list (N = 6), an appropriate array is dimensioned and the elements stored in this array, as Lines 90–120 indicate.

The aforementioned routine can be used in the final product as well as for trial runs of incomplete programs. In the developmental stages of a program, data may be continually altered as ideas are transformed into programming code. Final programs can use this routine to change data well after the program is written and used. Months later, when many of the coding techniques are forgotten, you need simply find and edit the data lines to tailor the program to your needs. Concern with redimensioning and altering of loops can be avoided because the program is developed so that it is easily adaptable to relevant updating.

ONERR GOTO statements can also be intentionally coded into the content of the program, making it a permanent part of a significant preplanned routine. One procedure that illustrates just such a use of an ONERR GOTO statement is

```
5  ONERR  GOTO 100
10  INPUT "ENTER A NUMBER";N
15  PRINT "THANK YOU"
20  END
100  PRINT "ENTER NUMBERS ONLY"
110  RESUME
```

Here we see the ONERR statement being used as an input check to make sure only numbers, not letters or other characters, are entered. A brief line by line explanation illustrates both the simplicity and power of this routine.

5 Sets the ONERR flag so that any programming error will transfer execution to Line 100.
10 Asks the user to enter a number and assigns this number to the variable N. If something other than a numerical entry is input, a ?REENTER message would normally appear. However, the ONERR GOTO statement traps this input error and executes the statements starting at Line 100 instead.
100 Prints a user friendly error message.
110 Sends command back to the beginning of the program line in which the error has occurred (in this case Line 10).

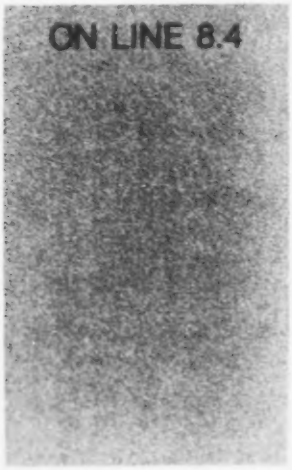
Remembering that the ONERR GOTO statement is in effect from time of execution until the program has completed its run, or a POKE 216,0 has been executed, one can easily appreciate the brevity it brings to routines.

Figure 8.2
If the string formed by concatenation is longer than 255 characters, the program goes to the subroutine that prints an error message and then returns to the main program for another input.

There are a few idiosyncrasies inherent in the Apple computer that allow for nonnumeric entries to bypass or escape the ONERR input check. If a single letter E or a group of digits along with the letter E is entered as input, it will be accepted by the computer as the E used in the exponential form of numeric expressions. It would be rejected, however, by an input routine utilizing the MID$ function,

as described in Section 4.5, because this routine compares the ASCII code values of the various keyboard characters entered. Two other entries that would not be trapped by the ONERR GOTO routine are the comma and the colon. If a colon or a comma is the first character of an INPUT response, Applesoft BASIC evaluates this response as zero when assigned to a numeric variable.

Overall, the ONERR GOTO statement is a vital programming statement to be used in the development, structure, and content of educational programs. Though underused because of its unfamiliarity to the beginning and intermediate programmer, the preceding examples have shown that the ONERR GOTO statement can be a powerful programming tool, one that can simplify the creation and improve the run of many educational programs.



1 List five different errors that result in the termination of a program.

2 Give two examples of how the ONERR GOTO statement can be used in the developmental stages of a program.

3 What statement cancels the ONERR GOTO error handling routine?

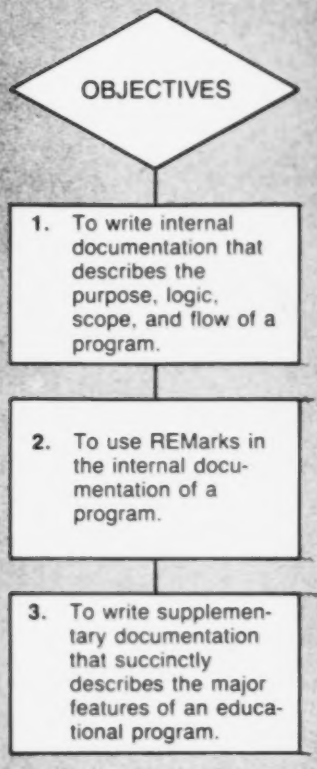
4 Explain the purpose of RESUME in an error handling routine.

5 Identify the two errors in the following program.

```
10  ONERR  GOTO 70
20 PRNT"FORGET IT!"
```

CHAPTER NINE

DOCUMENTATION


OBJECTIVES
1. To write internal documentation that describes the purpose, logic, scope, and flow of a program.
2. To use REMarks in the internal documentation of a program.
3. To write supplementary documentation that succinctly describes the major features of an educational program.

Proper documentation of a program makes it easier to read and understand. It is particularly valuable when a programmer at some future time modifies and enhances a program. Documentation is also of great assistance to someone, other than a program's author, who attempts to read and understand a program's logic and flow. This chapter identifies the kind of information requisite to good internal documentation and explains the use of REMark statements to include the documentation within a program. Section 9.3 discusses the preparation of concise supplementary documentation to accompany a program.

9.1 Why Document?

Appropriate documentation increases the effectiveness and usefulness of your educational programs. Documenting a BASIC program means including REMark statements to provide written descriptions that make the program easily readable, understandable, maintainable, and modifiable by you and, if necessary, other programmers. Such internal or self-documenting code should include meaningful identifiers as well as judiciously placed clarifying comments. For example, the following statement could be included in a program that computes the sum and average of a student's quiz grades for your class:

```
REM   S = SUM   A = AVERAGE
```

The REM statement informs the reader that the variable S will contain the sum of the quiz grades and that the variable A will contain the average of these grades.

In this program the REM statement in Line 200 informs the reader of the number of quiz grades to be processed.

```
200  REM  PROCESS 10 QUIZ GRADES
210  FOR C = 1 TO 10
220  INPUT N
230  NEXT C
```

Remember that documentation consists of REMarks that clarify the coding and internal logic of a program. Consequently, documenting your programs requires that you spend some time in writing and coding REM statements. Additional storage space in the computer will be required to hold a documented program. Thus some

well-intentioned, experienced programmers often tell beginning programmers to dispense with documenting their programs. Don't listen to or heed this poor advice. Only if you carefully document your programs will you and others be able to refer to them and make sense of them after they may have rested in your desk drawer for a few months or even a few years. Even more important, you and others will be able to easily revise and, in general, modify documented programs, which may be required for a new or different application. If your program is well documented, you and others need read only the REM statements to understand the purpose, logic, scope, and flow of the program.

In the past, before the need for complete and accurate documentation of programs was realized and stressed, even seasoned programmers often wrote undocumented programs. When they referred to them later, they discovered that they had no idea of the specific purpose, logic, or flow of the programs. Some programs were also spaghetti type in which program flow seemingly moved in almost random fashion. Imagine trying to modify such an undocumented mess. Often these programmers would move to another job or business, and other programmers would be asked to modify the program. Often it simply could not be done. Consequently, entirely new programs would have to be written. What a waste of a programmer's talent and effort! So be certain that you heed this advice: give careful consideration to documenting every one of your programs so that you and others will be able to debug, maintain, and modify them without spending an inordinate amount of time doing so.

---

1 List five reasons for documenting an educational computer program.

2 How are REMs used to document a program?

3 How does documentation relate to the structure of an educational computer program?

4 List the most important aspects of an educational computer program that should be documented internally.

9.2 What to Document

Experience is required to determine how much documentation is appropriate in your programs; the more complex the program, the more documentation required. Too much documentation will merely serve to clutter the program and mask its purpose, logic, and flow. On the other hand, too little documentation will not provide sufficient information or detail to permit the reader to understand, maintain, and modify the program as required.

Minimal documentation should include the programmer's name, date the program was coded, and the purpose of the program, all of which are placed in REM statements at the beginning of the program. In addition, REM statements that define key variables, describe statements that control the flow of the program, and clarify the steps listed in loops, subroutines, and algorithms should be judiciously placed throughout the program as required.

Table 9.1 lists some of the common items of internal documentation that you may wish to include in your programs.

Table 9.1
Documentation

Item	*Example*
Programmer's name	SUSIE SMART
Date	1 JANUARY, 2001
Program name	STATE CAPITALS
Program purpose	NAME THE CAPITALS OF 50 STATES
Subject	SOCIAL STUDIES
Intended audience	JUNIOR HIGH SCHOOL STUDENTS
Variables	S$ = STATE R = NUMBER OF CORRECT ANSWERS
Comment lines to explain control structures, loops, algorithms, and subroutines	SUBROUTINE TO COUNT NUMBER OF CORRECT ANSWERS

ON LINE 9.2

Write the appropriate documentation for an educational computer program you have written.

Format for Documentation 9.3

Internal documentation may be included in the form of REMs at the beginning of the program as shown in the example below. In this program the student is merely requested to enter the names of capitals of states in the United States. The REM statements provide appropriate initial documentation concerning information such as the programmer's name, date of coding, and purpose of program. Additional documentation could be placed throughout the remainder of the main program if desired.

```
10  REM  ************************************************************
14  REM  *   PROGRAM NAME: STATE CAPITALS                             *
18  REM  *                                                            *
22  REM  *   SUBJECT: SOCIAL STUDIES                                  *
26  REM  *                                                            *
30  REM  *   PROGRAM PURPOSE: NAMING CAPITALS OF STATES               *
34  REM  *                                                            *
38  REM  *   PROGRAM USE: DRILL AND PRACTICE                          *
42  REM  *                                                            *
46  REM  *   INTENDED AUDIENCE: JUNIOR HIGH SCHOOL STUDENTS           *
50  REM  *                                                            *
60  REM  *   PROGRAMMER: SUSIE SMART   1 JANUARY 2001                 *
64  REM  *                                                            *
68  REM  *   1550 UPBEAT STREET   SOMEWHERE, CALIFORNIA  91329        *
72  REM  *                                                            *
76  REM  *   COPYRIGHT 2001                                           *
84  REM  ************************************************************
```

Study Program A, which contains no internal documentation. Then compare it to the second version, Program B, that follows to convince yourself of the necessity and usefulness of documenting your educational programs.

```
A 100  PRINT "HOW MANY QUIZ GRADES WILL YOU BE ENTERING?"
  110  INPUT N
  120  PRINT "THERE ARE ";N;" QUIZ GRADES."
  130  LET S = 0
  140  FOR C = 1 TO N
  150  INPUT G
  160  LET S = S + G
  170  NEXT C
  190  LET A = S / N
  200  PRINT "SUM = ";S,"AVERAGE = ";A
  210  END
```

```
B 50  REM  PROGRAM TO COMPUTE THE SUM AND AVERAGE OF QUIZ GRADES
  60 :
  70  REM  INITIALIZE N
  80  INPUT N
  90  PRINT "THERE ARE ";N;" QUIZ GRADES."
 100  REM  INITIALIZE SUM
 110  LET S = 0
 120 :
 130  REM  PROCESS N QUIZ GRADES
 140  FOR C = 1 TO N
 150  REM  PROCESS EACH QUIZ GRADE
 160  INPUT G
 170  LET S = S + G
 180  NEXT C
 190  REM  COMPUTE AVERAGE AND PRINT RESULTS
 200  LET A = S / N
 210  PRINT "SUM = ";S,"AVERAGE = ";A
 220 :
 230  END
```

Program B offers much more information concerning not only the purpose of the program but also the purpose of the variables and routines. Actually one need only read Lines 50, 70, 100, 130, 150, and 190 to understand what the program does and how it does it. Note the : in Lines 60, 120, and 220 that serves to separate the modules in the listing, making the program more readable.

Program C, a third version, is also a structured program. It contains the same documenting REM statements as Program B. In addition, some statements are indented to various levels to improve the readability and clarity. Note the use of colons to force the Apple to maintain proper indentation for each line.

```
C 100  REM  PROGRAM TO COMPUTE SUM
  110  REM  AND AVERAGE OF QUIZ
  120  REM  GRADES
  130 :
  140  REM  INITIALIZE N
  150 ::::: INPUT N
  160 ::::: PRINT "THERE ARE ";N;
  170 ::::: PRINT " QUIZ GRADES"
  180 ::::: PRINT
  200  REM  INITIALIZE SUM
  210 ::::: LET S = 0
  220 :
  230  REM  PROCESS N QUIZ GRADES
  240 ::::: FOR C = 1 TO N
  250 :::::::::: REM  PROCESS EACH
  260 :::::::::: REM  QUIZ GRADE
  270 :::::::::: INPUT G
```

```
280 :::::::::: LET S = S + G
290 ::::: NEXT C
300 :
310  REM  COMPUTE AND PRINT
320  REM  SUM AND AVERAGE
330  REM  OF QUIZ GRADES
340 ::::: LET A = S / N
350 ::::: PRINT "SUM = ";S
360 ::::: PRINT
370 ::::: PRINT "AVERAGE = ";A
380 :
390 ::::: END
```

If you were to type this into the Apple with just spaces, instead of colons, it would appear, as you were typing, that the Apple is maintaining the indentations. However, when you list the completed program, you will see that the Apple left justifies all of the program statements. Colons must be used on the Apple to maintain the indentations. The indenting format is an implicit form of internal documentation because it helps the reader to quickly identify the various control structures and loops, to determine their purpose, and to modify the program as required. At present the concept of structure as related to programming is relatively new. For this reason different programmers have formed their own definition of structure. In general, structured programming is a method of designing and writing programs so that they can be easily read, understood, maintained, and modified.

Figure 9.1
Internal documentation for Animal Heartbeats.

```
10 REM  ****************************************
20 REM  ****************************************
30 REM  PROGRAM: ANIMAL HEARTBEATS
40 REM  SUBJECT: LIFE SCIENCE
50 REM  PURPOSE: TO COMPARE THE RELATIVE SIZE OF
60 REM  ANIMALS AND THEIR HEARTBEAT RATES.
70 REM  PROGRAM USE: SIMULATION
80 REM  INTENDED AUDIENCE: LIFE SCIENCE STUDENTS
90 REM  PROGRAMMER: C. HEINLER  4/15/86
100 REM  COPYRIGHT 1986
110 REM  ****************************************
120 REM  ****************************************
130 :
140 REM  VARIABLES
150 H$ = "HORSE":C$ = "CAT":P$ = "PIG":G$ ="GUINEA PIG":E$ = "ELEPHANT"
    :D$ = "GUINEA PIG"
```

(continued)

160 REM BEEP SPEAKER FOR SOUND OF HEART BEAT
170 REM
INTEGER (1-5)
180 REM R = NUMBER OF HEARTBEATS SOUNDED X = VALUE FOR DELAY LOOP TO T
THE RATE OF HEARTBEATS

Figure 9.1
(Continued)

If your educational program places specific requirements on the computer or operating system you may wish to document these in the program. Is a color monitor required or will a black-and-white television set suffice? Will sound be generated by the computer? Are paddles or a joystick needed for input?

Up to this point we have discussed the need to document within the program tangible items, such as the programmer's name and the purpose of the program. But there may also be a need to provide written external documentation, also called supplementary documentation, for the teacher or student concerning other items, such as content of the program, background information necessary to permit successful use of the program, and additional information concerning possible modes of interaction between the student and the program: for example, the use of a worksheet with a program, like The Number Machine worksheet in Appendix B. This supplementary documentation might be only a single page, like the Documentation Guide in Figure 9.2, or it may be more extensive and take the form of a teacher's guide that provides assistance to the teacher using the program.

Assume a tenth grade biology class is using a computer simulation program called Dirty Water to model the effects of certain variables on the quality of a water resource. Students can use the program alone or in small groups. The program allows the student to designate a body of water (pond, lake, stream) and to control the temperature of the water, the amount and type of waste added, the rate of addition, and the type of waste treatment. After accepting values for these variables, the computer prints a chart and graph showing the oxygen content and the waste content of the water for each simulated day until the system reaches equilibrium. Fish will begin to die if the oxygen content drops below 5 parts per million (ppm). As the student continues to rerun the program, the interplay of the variables and their contribution to water quality become evident.

The Dirty Water program would certainly contain extensive internal documentation, including a comprehensive listing of the purpose and definitions of all key variables. In addition, there could be substantial external documentation. A student workbook could

Date: _____

Programmer's Name: _____ Phone: _____

Address: _____

Program Name: _____ Date of Coding: _____

Program Purpose and Description: _____

_____

Program Use (Drill and practice, game, simulation, tutorial etc.):

_____

Intended Audience: (Grade level / age level): _____

Critical Prerequisite Concepts or Skills: _____

_____

Price: _____ Copyright Information: _____

System Requirements

Computer: _____ Language: _____

Memory Required: _____ Color (Y/N): __ Sound (Y/N): __

Input Mode: Output Mode:

Keyboard (Y/N) _____ Monitor (Y/N) _____

Joystick (Y/N) _____ Printer (Y/N) _____

Paddles (Y/N) _____ Disk Drive (Y/N) _____

Mouse (Y/N) _____ Plotter (Y/N) _____

Other _____ Other _____

Recommended or Supplementary Material: _____

_____

Figure 9.2
Documentation Guide

suggest to the student certain preliminary investigations using specified values for the various variables. This workbook could also define and explain important terms such as ppm and primary, secondary, and tertiary water treatment.

A teacher's guide may be developed that explains how to use the Dirty Water program in a teaching unit. This guide could list the teaching objectives, explain the rationale for the Dirty Water

model, provide the teacher with information about the concepts that students must understand before using the program, describe the major goals of the unit, and provide follow-up questions to be used during classroom discussions of water pollution. The teacher's guide could also provide the teacher with necessary background information about the scientific principles involved: for example, chemical and physical changes, solubility of oxygen and waste in water, types of human and industrial waste, and methods of improving the quality of water. References to journal articles and films related to the program, a list of possible student projects, a clear description of the mathematical equations and constants used in the Dirty Water model, and the assumptions under which the model operates could also be included in the teacher's guide.

Documentation is an essential part of every good educational program. Appropriate internal documentation in the form of REM statements will professionalize your program and help you revise the program over time. From your own experience you probably know that the frustrations that sometimes occur when using commercial software are often associated with the lack of specific and precise external documentation explaining clearly and accurately how to run and use the software. Simple and succinct external documentation for your educational computer programs will help avoid this problem by assisting students and teachers in understanding the objectives and purposes of the program and how to use it successfully with optimum ease and maximum benefit.

---

1 Add REM statements to clarify the purpose of the following program:

```
10 PRINT "ENTER TWO POSITIVE INTEGERS"
20 INPUT N1,N2
30 IF N1 > N2 THEN  PRINT N1;" IS LARGER THAN ";N2: GOTO 60
40 IF N2 > N1 THEN  PRINT N2;" IS LARGER THAN ";N1: GOTO 60
50 PRINT N1;" IS EQUAL TO ";N2
60 END
```

2 What is external documentation, and what is its purpose?

3 Use colons for indenting lines to improve your documentation in the preceding exercise.

4 Complete the documentation guide shown on page 241 for each of the programs ANIMAL HEARTBEATS, END PUNCTUATION, and THE NUMBER MACHINE.

```
10  REM  THE NUMBER MACHINE - WRITING WHOLE NUMBERS IN WORDS
30  REM  A PROGRAM THAT ALLOWS YOU TO ENTER A NUMBER
40  REM  AND SEE IT WRITTEN IN LITERAL FORM
80 :
90 :
100  REM  IDENTIFY VARIABLES
110  REM  G$,G = GRADE LEVEL :: N$ = NUMBER TO BE CONVERTED TO WORDS
120  REM  Z$ = NUMBER WITH COMMAS ADDED::W$ = NUMBER CONVERTED TO WORDS
130  REM   V(1), V(2), V(3) = VALUE OF HUNDREDS', TENS' & UNITS' DIGITS
140  REM  NG$ = GROUP OF 3 NUMBERS :: WG$ = WORD GROUP OF 3 NUMBERS
150  REM NC = THE AMOUNT OF COMMAS IN THE NUMBER :: S$ = SOUND OPTION
160 :
170 :
180  REM .....READ DATA AND INITIALIZE VARIABLES.....
190 :
200 :
210  REM   READ UNITS' PLACE (AND HUNDREDS' PLACE)
220  FOR I = 1 TO 9
230  READ U$(I)
240  NEXT I
250  DATA  ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE
260 :
270 :
280  REM    READ TENS' PLACE
290  FOR I = 2 TO 9
300  READ T$(I)
310  NEXT I
320  DATA TWENTY ,THIRTY ,FORTY ,FIFTY ,SIXTY ,SEVENTY ,EIGHTY ,NINETY
330 :
340 :
350  REM  READ THE 'TEENS'
360  FOR I = 0 TO 9
370  READ TE$(I)
380  NEXT I
390  DATA  TEN,ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN,SIXTEEN,SEVENTEEN
    ,EIGHTEEN,NINETEEN
400 :
410 :
420  REM  READ THE 'BIG' NUMBERS
430  FOR I = 1 TO 9
440  READ B$(I)
450  NEXT I
460  DATA   " THOUSAND "," MILLION "," BILLION "," TRILLION "," QUADRILL
   ION "," QUINTILLION "," SEXTILLION "," SEPTILLION "," OCTILLION "
470 :
480 :
490  REM .....SET-UP AND INTRODUCTION.....
500 :
```

```
510 :
520  HOME
530 W$ = "THE NUMBER MACHINE"
540  GOSUB 2000: REM CREATE CHARACTER GRAPHIC OF NUMBER MACHINE
550  POKE 34,15: REM  KEEP MACHINE ON SCREEN WHEN HOME IS EXECUTED
560  FOR P = 1 TO 1000: NEXT P
570  GOSUB 3000: GOSUB 3100: REM PAD AND MOVE TITLE
580  VTAB 16: PRINT  TAB( 5);"HELLO, I'M THE NUMBER MACHINE"
590  PRINT : PRINT "PLEASE TYPE YOUR NAME AND PRESS RETURN"
600  PRINT : INPUT NA$
610  VTAB 22: PRINT "HI ";NA$;", WHAT GRADE ARE YOU IN (1-12)";
620  INPUT G$
630 G =  VAL (G$)
640  IF G < 1 OR G > 12 THEN  HOME : PRINT "PLEASE ENTER A GRADE BETWEEN
     1 AND 12": GOTO 610
650  VTAB 24: PRINT "DO YOU WANT SOUND (Y/N)?";
660  GET S$
670 :
680 :
690  REM .....MAIN PROGRAM.....
700 :
710 :
720  REM   CATCH 'STRING TOO LONG' ERROR IF PHRASE IS > 255 CHARACTERS
730  ONERR  GOTO 5000
740 :
750  HOME
760  PRINT  TAB( 5);"ENTER A NUMBER AND WATCH ME GO!"
770 :
780  REM INIT HOLDING & COUNTING VARIABLES EACH TIME PROGRAM IS RUN
790 W$ = "":Z$ = "":NC = 0:NN$ = "":C = 0
800 :
810  PRINT : INPUT "NUMBER: ";N$
820 :
830  REM  CHECK FOR TRAILING SPACES IN INPUT
840  IF  RIGHT$ (N$,1) = " " THEN N$ =  LEFT$ (N$, LEN (N$) - 1): GOTO 8
    40
850 :
860  REM  CHECK THAT ONLY NUMBERS ARE ENTERED
870  FOR I = 1 TO  LEN (N$)
880  IF  MID$ (N$,I,1) >  = "0" AND  MID$ (N$,I,1) <  = "9" THEN  NEXT I

890  IF I =  LEN (N$) + 1 THEN 950
900  HOME
910  PRINT  TAB( 10);: INVERSE : PRINT "ENTER NUMBERS ONLY": NORMAL
920  GOTO 810
930 :
940  REM LIMIT SIZE OF NUMBER FOR LOWER GRADES
950  IF G < 6 AND  LEN (N$) > G + 1 THEN  HOME : PRINT "PLEASE ENTER A S
    MALLER NUMBER": GOTO 810
960 :
970  GOSUB 4000: REM PUT COMMAS IN NUMBER
```

```
980 :
990  REM  CENTER AND HIGHLIGHT NUMBER ON SCREEN
1000  HOME
1010  PRINT  TAB( (40 -  LEN (Z$)) / 2);: INVERSE : PRINT " ";Z$;" "
1020  VTAB 8: HTAB 11: FLASH : PRINT "  T H I N K I N G  ": NORMAL
1030 :
1040  REM   DELAY LOOP WITH SOUND TO KEEP 'THINKING' ON THE SCREEN
1050  FOR P = 1 TO 300 * (NC + 1)
1060  IF S$ = "Y" THEN S =  PEEK ( - 16336)
1070  NEXT P
1080 :
1090  REM PAD NO. WITH ZEROS SO EACH GROUP BETWEEN COMMAS HAS 3 DIGITS
1100  IF  LEN (Z$) < 3 THEN Z$ = "0" + Z$: GOTO 1100
1110  IF  LEN (Z$) > 3 AND  MID$ (Z$,4,1) <  > "," THEN Z$ = "0" + Z$: GOTO
    1110
1120 :
1130  REM CONVERT GROUPS OF 3 DIGITS TO WORDS
1140  FOR I = 1 TO  LEN (Z$) STEP 4
1150 NG$ =  MID$ (Z$,I,3)
1160 H$ = " HUNDRED "
1170  IF  LEFT$ (NG$,1) = "0" THEN H$ = ""
1180  FOR J = 1 TO 3
1190 V(J) =  VAL ( MID$ (NG$,J,1))
1200  NEXT J
1210  IF V(2) = 1 THEN T$(V(2)) = TE$(V(3)):V(3) = 0
1220 WG$ = U$(V(1)) + H$ + T$(V(2)) + U$(V(3))
1230  IF WG$ <  > "" THEN WG$ = WG$ + B$(NC)
1240 W$ = W$ + WG$
1250 NC = NC - 1
1260  NEXT I
1270  IF W$ = "" THEN W$ = "ZERO"
1280 :
1290  REM  TELL USER THAT NUMBER MACHINE IS READY TO DISPLAY NUMBER
1300  VTAB 24: HTAB 10: PRINT "PRESS ANY KEY TO START";
1310  VTAB 8: HTAB 11: PRINT "      .:READY:.      "
1320  IF S$ = "Y" THEN S =  PEEK ( - 16336)
1330  VTAB 8: HTAB 12: PRINT "                      "
1340  IF  PEEK (49152) < 128 THEN 1310: REM CHECK FOR KEYPRESS
1350 :
1360  REM CLEAR KEYBOARD STROBE & MESSAGE ON BOTTOM OF SCREEN
1370  POKE  - 16368,0: VTAB 23: CALL  - 958
1380 :
1390  REM MOVE AND DISPLAY NUMBER IN LITERAL FORM
1400  GOSUB 3000: GOSUB 3100
1410  VTAB 17: HTAB 22: PRINT W$
1420 :
1430  REM  QUIT OPTION
1440  VTAB 24: HTAB 10: PRINT "TRY AGAIN (Y/N)?";
1450  GET K$
1460  IF K$ = "Y" THEN 750
1470  IF K$ <  > "N" THEN 1450
```

```
1480  TEXT : HOME : END
1940 :
1950 :
1960  REM .....SUBROUTINES START HERE.....
1970 :
1980 :
1990  REM  CHARACTER GRAPHIC OF THE NUMBER MACHINE
2000  VTAB 2: HTAB 10: PRINT "//////////////////////////////"
2010  VTAB 3: HTAB 9: PRINT "//////////////////////////////"
2020  VTAB 4: HTAB 8: PRINT "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!///"
2030  VTAB 5: HTAB 8: PRINT "!..........................///"
2040  VTAB 6: HTAB 8: PRINT "!::::::::::::::::::::::::::::///"
2050  VTAB 7: HTAB 8: PRINT "!:.                          .:///"
2060  VTAB 8: HTAB 8: PRINT "!:.                          .:///"
2070  VTAB 9: HTAB 8: PRINT "!:..........................:///"
2080  VTAB 10: HTAB 8: PRINT "!::::::::::::::::::::::::::::///"
2090  VTAB 11: HTAB 8: PRINT "!..........................///"
2100  VTAB 12: HTAB 8: PRINT "!..........................//"
2110  VTAB 13: HTAB 8: PRINT "!_____________________________/"
2120  RETURN
2970 :
2980 :
2990  REM   PACK PHRASE WITH SPACES BEFORE AND AFTER
3000  FOR I = 1 TO 19
3010 W$ = " " + W$
3020  NEXT I
3030 W$ = W$ + " "
3040  RETURN
3070 :
3080 :
3090  REM   ROUTINE TO MOVE PHRASE
3100  FOR I = 1 TO  LEN (W$)
3110  VTAB 8: HTAB 11: PRINT  MID$ (W$,I,19)
3120  FOR P = 1 TO 150: NEXT P
3130  NEXT I
3140  RETURN
3970 :
3980 :
3990  REM   ROUTINE TO INSERT COMMAS IN NUMBER
4000  FOR I =  LEN (N$) TO 1 STEP  - 1
4010 M$ =  MID$ (N$,I,1)
4020 C = C + 1
4030  IF C = 3 AND I <  > 1 THEN M$ = M$ + ",":C = 0:NC = NC + 1
4040 NN$ = NN$ + M$
4050  NEXT I
4060  FOR I =  LEN (NN$) TO 1 STEP  - 1
4070 Z$ = Z$ +  MID$ (NN$,I,1)
4080  NEXT I
4090  RETURN
4970 :
4980 :
4990  REM   ONERR ROUTINE TO CATCH 'STRING TOO LONG' ERROR
5000  HOME
5010  PRINT "THE NUMBER YOU ENTERED IS TOO LARGE"
5020  VTAB 8: HTAB 11: PRINT "                    "
5030  VTAB 17: GOTO 790
```

The Number Machine Worksheet

Appendix B

Example of one kind of worksheet to be used with THE NUMBER MACHINE (Appendix A). Worksheets similar to this one may be developed for other ranges of numbers.

THE NUMBER MACHINE
"One Thousand"

Name: _____

Objective: To practice writing numbers as words.

The Number Machine makes a computer print the words for a number.

Try this example:

Type 1000 and press Return.
See ONE THOUSAND printed on the computer screen.
ONE THOUSAND is 1000 in words.

A. Type each number into The Number Machine and press Return. Write the words for the numbers in the spaces at the right.

NUMBER	WORDS
1025	_____
1150	_____
1317	_____

B. 1. In the space below, write the words for each number. Do NOT use The Number Machine.

NUMBER	WORDS
1279	_____
1304	_____
1612	_____

2. Enter each number above into The Number Machine to check your work. Change your answer if it does not agree with the words printed on the computer screen.

C. 1. Write the words for 3 numbers that begin with "One Thousand."

_____

_____

_____

2. Now, write the number for each of these words.

_____

_____

_____

3. Type each number into The Number Machine and check your work. Correct any wrong words.

Appendix C End Punctuation

```
10  REM  ********************************************************
20  REM  ********************************************************
30  REM  PROGRAM: END PUNCTUATION
40  REM  SUBJECT: ENGLISH GRAMMAR
50  REM  PURPOSE: PUNCTUATING THE END OF A SENTENCE
60  REM  PROGRAM USE: TUTORIAL
70  REM  INTENDED AUDIENCE: UPPER ELEMENTARY
80  REM  PROGRAMMERS: J. CUNNINGHAM AND C. HEIMLER   6/28/86
90  REM  COPYRIGHT: JULY, 1986
100  REM  ********************************************************
110  REM  ********************************************************
120 :
130 :
140  REM  IDENTIFY VARIABLES
150  REM  K$ = USER'S INPUT TO MENU AND SPACE BAR ROUTINE
160  REM  N = 10 RANDOM NUMBERS IN THE RANGE 1 TO 20
170  REM  R = NUMBER OF CORRECT ANSWERS TO QUIZ
180  REM  L = NUMBER OF CHARACTERS IN S$
190  REM  P1 = COLUMN POSITION TO START PRINTING S$
200  REM  S$ = QUIZ SENTENCES::M$ = USER'S INPUT TO QUIZ QUESTIONS
210  REM  P$ = QUIZ PUNCTUATION MARK::FB$ = QUIZ FEEDBACK
220  REM  EV$ = FEEDBACK FOR EVALUATION::KB = KEYBOARD CONTROL
225  SPEED= 255
230  REM  PRINT BORDER
240  HOME
250  FOR I = 1 TO 2
260  PRINT "?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?";
270  NEXT I
280  FOR I = 1 TO 18
290  PRINT "?!                                                  !?";
300  NEXT I
310  FOR I = 1 TO 2
320  PRINT "?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?";
330  NEXT I
340  REM  PRINT TITLE
350  VTAB 8: HTAB 11: PRINT "......................."
360  VTAB 9: HTAB 11: PRINT ".                     ."
370  VTAB 10: HTAB 11: PRINT ".                     ."
380  VTAB 11: HTAB 11: PRINT ".                     ."
390  VTAB 12: HTAB 11: PRINT ".                     ."
400  VTAB 13: HTAB 11: PRINT "......................."
410  SPEED= 25
420  VTAB 11: HTAB 14: PRINT "END PUNCTUATION"
430  SPEED= 255
440  FOR P = 1 TO 3500: NEXT P
450  HOME
460 :
```

```
470  GOSUB 2590: REM  SEND TO SUBROUTINE FOR ASSIGNING SENTENCES, PUNCTU
    ATION MARKS, AND FEEDBACK TO VARIABLES
480 :
490  REM  PRINT GREETING AND OBJECTIVE
500  VTAB 8
510  SPEED= 50
520  PRINT "WELCOME TO YOUR FRIENDLY ENGLISH"
530  PRINT
540  PRINT "TEACHER. THIS IS A LESSON ON PUNCTUATING"
550  PRINT "THE END OF AN ENGLISH SENTENCE."
560  GOSUB 2480: REM  SPACE BAR USER CONTROL
570  HOME
580  VTAB 8
590  PRINT "OBJECTIVE: AFTER COMPLETING THIS LESSON"
600  PRINT
610  PRINT "YOU WILL BE ABLE TO CORRECTLY PUNCTUATE"
620  PRINT
630  PRINT "THE END OF A SENTENCE."
640  SPEED= 255
650  GOSUB 2480
660  HOME : VTAB 4
670  PRINT  TAB( 14)"*** MENU ***"
680  PRINT : PRINT
690  PRINT  TAB( 14)"A. TUTORIAL"
700  PRINT : PRINT
710  PRINT  TAB( 14)"B. QUIZ"
720  PRINT : PRINT
730  PRINT  TAB( 14)"C. QUIT"
740  PRINT : PRINT : PRINT
750  PRINT  TAB( 12)"PRESS A, B, OR C."
760  GET K$
770  IF K$ = "A" THEN 830
780  IF K$ = "B" THEN 2010
790  IF K$ = "C" THEN 2450
800  IF K$ <  > "A" OR K$ <  > "B" OR K$ <  > "C" THEN  HOME : GOTO 660
810 :
820  REM  TUTORIAL ROUTINE
830  HOME
840  PRINT  TAB( 16)"TUTORIAL"
850  PRINT  TAB( 16)"--------"
860  POKE 34,2
870  VTAB 5
880  PRINT "THERE ARE THREE PUNCTUATION MARKS THAT "
890  PRINT "CAN BE USED AT THE END OF A SENTENCE."
900  VTAB 12
910  PRINT  TAB( 5);"PERIOD"; TAB( 25);"."
920  PRINT
930  PRINT  TAB( 5);"QUESTION MARK"; TAB( 25);"?"
940  PRINT
950  PRINT  TAB( 5);"EXCLAMATION MARK"; TAB( 25);"!"
```

```
960  GOSUB 2480
970  HOME
980  VTAB 4: HTAB 10: INVERSE : PRINT "DECLARATIVE SENTENCE": NORMAL
990  VTAB 8
1000  PRINT "A DECLARATIVE SENTENCE MAKES A STATEMENT"
1010  PRINT "ABOUT SOMEONE OR SOMETHING."
1020  PRINT
1030  PRINT "USE A PERIOD AT THE END OF A DECLARATIVE"
1040  PRINT "SENTENCE."
1050  GOSUB 2480
1060  HOME
1070  VTAB 4: PRINT "EXAMPLES OF DECLARATIVE SENTENCES:"
1080  VTAB 8
1090  SPEED= 50
1100  PRINT  TAB( 5)"A WHALE IS THE LARGEST MAMMAL"
1110  VTAB 8: HTAB 34: FLASH : PRINT ".": NORMAL
1120  VTAB 10
1130  PRINT  TAB( 5)"DIAMONDS ARE VERY HARD"
1140  VTAB 10: HTAB 27: FLASH : PRINT ".": NORMAL
1150  VTAB 12
1160  PRINT  TAB( 5)"THE DOG LIKES TO EAT MEAT"
1170  VTAB 12: HTAB 30: FLASH : PRINT ".": NORMAL
1180  VTAB 14
1190  PRINT  TAB( 5)"A ROOSTER CROWED THIS MORNING"
1200  VTAB 14: HTAB 34: FLASH : PRINT ".": NORMAL
1210  SPEED= 255
1220  GOSUB 2480
1230  HOME
1240  VTAB 4: HTAB 10: INVERSE : PRINT "IMPERATIVE SENTENCE": NORMAL
1250  VTAB 8
1260  PRINT "AN IMPERATIVE SENTENCE STATES A REQUEST OR COMMAND."
1270  PRINT
1280  PRINT "USE A PERIOD AT THE END OF AN IMPERATIVESENTENCE."
1290  GOSUB 2480
1300  HOME
1310  VTAB 4: PRINT "EXAMPLES OF IMPERATIVE SENTENCES:"
1320  VTAB 8
1330  SPEED= 50
1340  PRINT  TAB( 5)"PLEASE CLOSE THE WINDOW"
1350  VTAB 8: HTAB 28: FLASH : PRINT ".": NORMAL
1360  VTAB 10
1370  PRINT  TAB( 5)"GIVE ME THE MONEY"
1380  VTAB 10: HTAB 22: FLASH : PRINT ".": NORMAL
1390  VTAB 12
1400  PRINT  TAB( 5)"PLEASE RETURN THE LIBRARY BOOK"
1410  VTAB 12: HTAB 35: FLASH : PRINT ".": NORMAL
1420  VTAB 14
1430  PRINT  TAB( 5)"WARM UP BEFORE YOU PLAY THE GAME"
1440  VTAB 14: HTAB 37: FLASH : PRINT ".": NORMAL
1450  SPEED= 255
1460  GOSUB 2480
```

```
1470  HOME
1480  VTAB 4: HTAB 9: INVERSE : PRINT "INTERROGATIVE SENTENCE": NORMAL
1490  VTAB 8
1500  PRINT "AN INTERROGATIVE SENTENCE REQUESTS AN "
1510  PRINT "ANSWER TO A QUESTION."
1520  PRINT
1530  PRINT "USE A QUESTION MARK AT THE END OF AN "
1540  PRINT "INTERROGATIVE SENTENCE."
1550  GOSUB 2480
1560  HOME
1570  VTAB 4: PRINT "EXAMPLES OF INTERROGATIVE SENTENCES:"
1580  SPEED= 50
1590  VTAB 8
1600  PRINT  TAB( 5)"WHAT IS YOUR NAME"
1610  VTAB 8: HTAB 22: FLASH : PRINT "?": NORMAL
1620  VTAB 10
1630  PRINT  TAB( 5)"HOW FAST CAN YOU RUN"
1640  VTAB 10: HTAB 25: FLASH : PRINT "?": NORMAL
1650  VTAB 12
1660  PRINT  TAB( 5)"DO YOU LIKE ICE CREAM"
1670  VTAB 12: HTAB 26: FLASH : PRINT "?": NORMAL
1680  VTAB 14
1690  PRINT  TAB( 5)"WHEN WILL WE MEET AGAIN"
1700  VTAB 14: HTAB 28: FLASH : PRINT "?": NORMAL
1710  SPEED= 255
1720  GOSUB 2480
1730  HOME
1740  VTAB 4: HTAB 10: INVERSE : PRINT "EXCLAMATORY SENTENCE": NORMAL
1750  VTAB 8
1760  PRINT "AN EXCLAMATORY SENTENCE EXPRESSES STRONGFEELINGS OR EMOTION
    S."
1770  PRINT
1780  PRINT "USE AN EXCLAMATION MARK (!) AT THE END  OF AN EXCLAMATORY S
    ENTENCE."
1790  GOSUB 2480
1800  HOME
1810  VTAB 4: PRINT "EXAMPLES OF EXCLAMATORY SENTENCES:"
1820  VTAB 8
1830  SPEED= 50
1840  PRINT  TAB( 5)"I LOVE LIFE"
1850  VTAB 8: HTAB 16: FLASH : PRINT "!": NORMAL
1860  VTAB 10
1870  PRINT  TAB( 5)"YOU MUST LEAVE NOW"
1880  VTAB 10: HTAB 23: FLASH : PRINT "!": NORMAL
1890  VTAB 12
1900  PRINT  TAB( 5)"WE WON THE GAME"
1910  VTAB 12: HTAB 20: FLASH : PRINT "!": NORMAL
1920  VTAB 14
1930  PRINT  TAB( 5)"WHAT A TIME WE HAD"
1940  VTAB 14: HTAB 23: FLASH : PRINT "!": NORMAL
1950  SPEED= 255
```

```
1960  GOSUB 2480
1970  TEXT : REM  CLEAR SCREEN POKE
1980  GOTO 660: REM  BACK TO MENU
1990 :
2000  REM  GENERATE AND SAVE 10 DIFFERENT RANDOM NUMBERS FROM 1 TO 20
2010  FOR I = 1 TO 10
2020  LET N(I) =  INT (20 *  RND (1) + 1)
2030  FOR J = 0 TO I - 1
2040  IF N(I) = N(J) THEN 2020
2050  NEXT J
2060  NEXT I
2070 :
2080  REM  PUNCTUATION QUIZ
2090  LET R = 0
2100  HOME
2110  HTAB 14: INVERSE : PRINT "*** QUIZ ***": NORMAL
2120  VTAB 4: PRINT "TYPE THE CORRECT PUNCTUATION MARK FOR   THE END OF
    THE SENTENCE."
2130  PRINT "----------------------------------------"
2140  POKE 34,2
2150  FOR I = 1 TO 10
2160  HOME
2170  VTAB 4: PRINT "TYPE THE CORRECT PUNCTUATION MARK FOR   THE END OF
    THE SENTENCE."
2180  PRINT "----------------------------------------"
2190  LET L =  LEN (S$(N(I)))
2200  LET P1 = (40 - L) / 2
2210  VTAB 12: HTAB P1: PRINT S$(N(I))
2220  VTAB 12: HTAB P1 + L: GET M$
2230  VTAB 12: HTAB P1 + L: PRINT M$
2240  IF M$ <  > "." AND M$ <  > "?"AND M$ <  > "!"THEN  SPEED= 50: VTAB
    17: HTAB 1: PRINT "PLEASE TYPE A PERIOD, QUESTION MARK OR   EXCLAMAT
    ION MARK.": SPEED= 255: GOSUB 2480: HOME : GOTO 2210
2250  PRINT : PRINT
2260  IF M$ = P$(N(I)) THEN  VTAB 17: PRINT "YOUR PUNCTUATION MARK IS CO
    RRECT.": LET R = R + 1: GOSUB 2480: HOME : GOTO 2380
2270  IF M$ = "." AND P$(N(I)) = "!" THEN  VTAB 15: PRINT "YES, A PERIOD
     COULD BE USED.  BUT AN   EXCLAMATION MARK IS USED IF THE SENTENCE E
    XPRESSES A STRONG FEELING OR EMOTION.": LET R = R + 1: GOSUB 2480:
    GOTO 2380
2280  VTAB 18: PRINT "YOUR PUNCTUATION MARK IS NOT CORRECT."
2290  GOSUB 2480
2300  HOME
2310  VTAB 7: PRINT "LOOK AT THE SENTENCE AGAIN."
2320  VTAB 10: HTAB P1: INVERSE : PRINT S$(N(I)): NORMAL
2330  VTAB 14: PRINT FB$(N(I))
2340  PRINT
2350  PRINT "THE SENTENCE SHOULD END WITH A "P$(N(I))
2360  VTAB 10: HTAB P1 + L: FLASH : PRINT ;P$(N(I)): NORMAL
2370  GOSUB 2480
2380  NEXT I
```

```
2390  TEXT : HOME
2400  VTAB 2: HTAB 14: FLASH : PRINT "END OF QUIZ": NORMAL
2410  VTAB 8: HTAB 6: PRINT "YOUR SCORE: ";R;" OUT OF 10"
2420  GOSUB 2530
2430  VTAB 14: HTAB 6: PRINT "EVALUATION: ";EV$
2440  GOSUB 2480: GOTO 660: REM  BACK TO MAIN MENU
2450  END
2460 :
2470  REM  SUBROUTINE USING SPACE BAR FOR USER CONTROL
2480  VTAB 20: HTAB 5: PRINT "(PRESS SPACE BAR TO CONTINUE)"
2490 KB =  PEEK (49152): IF KB <  > 160 THEN 2490
2500  POKE 49168,0
2510  RETURN
2515 :
2520  REM  QUIZ EVALUATION FEEDBACK
2530  IF R >  = 9 THEN EV$ = "EXCELLENT!"
2540  IF R = 8 THEN EV$ = "VERY GOOD"
2550  IF R = 7 THEN EV$ = "SATISFACTORY"
2560  IF R <  = 6 THEN EV$ = "MORE STUDY NEEDED"
2570  RETURN
2580 :
2590  REM  ASSIGNING SENTENCES, PUNCTUATION, AND FEEDBACK TO SUBSCRIPTED
    VARIABLES
2600  DIM S$(20),P$(20),FB$(20)
2610  FOR I = 1 TO 20
2620  READ S$(I),P$(I),FB$(I)
2630  NEXT I
2640  RETURN
2650 :
2660  REM  DATA STATEMENTS - SENTENCES, PUNCTUATION, FEEDBACK
2670  REM  DECLARATIVE SENTENCES
2680  DATA  I PLAYED A VIDEO GAME,.,THE STATEMENT IS ABOUT A GAME.
2690  DATA  ALASKA IS ONE OF THE FIFTY STATES,.,THE STATEMENT IS ABOUT A
    LASKA.
2700  DATA  GASOLINE COMES FROM PETROLEUM,.,THIS STATEMENT IS ABOUT GASO
    LINE.
2710  DATA  HIS DOG ATE THE BONE,.,THE SENTENCE IS A STATEMENT ABOUT A D
    OG
2720  DATA  READING IS IMPORTANT,.,THIS IS A STATEMENT ABOUT READING.
2730  REM  INTERROGATIVE SENTENCES
2740  DATA  MAY I BORROW A PENCIL,?,A PERSON WANTS A PENCIL.
2750  DATA  HOW TALL ARE YOU,?,A PERSON IS ASKED HIS/HER HEIGHT.
2760  DATA  WHAT IS YOUR BEST FRIEND'S NAME,?,GIVING A NAME WILL ANSWER
    THE QUESTION.
2770  DATA  WHAT TIME IS IT,?,A PERSON WANTS TO KNOW THE TIME.
2780  DATA  DO YOU ENJOY SCHOOL,?,A YES OR NO WILL ANSWER THE QUESTION.
2790  REM  IMPERATIVE SENTENCES
2800  DATA  PLEASE CLOSE THE DOOR,.,A REQUEST IS MADE.
2810  DATA  TURN RIGHT AT THE CORNER,.,A COMMAND IS GIVEN.
2820  DATA  PLEASE GIVE ME A PENCIL,.,THE SENTENCE MAKES A REQUEST.
2830  DATA  DO YOUR HOMEWORK,.,THE SENTENCE GIVES A COMMAND.
```

```
2840  DATA  PLEASE BE QUIET IN THE LIBRARY,.,THE SENTENCE MAKES A REQUES
    T.
2850  REM  EXCLAMATORY SENTENCES
2860  DATA  I AM VERY HAPPY,!,THE SENTENCE EXPRESSES A STRONG FEELING.
2870  DATA  I WON,!,WINNING OFTEN INVOLVES A STRONG EMOTION.
2880  DATA  HE IS A FOOL,!,THE SENTENCE EXPRESSES STRONG EMOTION.
2890  DATA  HOW I HATED HIM,!,HATE IS A STRONG EMOTION.
2900  DATA  WHAT A GOOD TIME WE HAD,!,THIS CAN BE AN EMOTIONAL STATEMENT
    .
2910  RETURN
```

Animal Heartbeats *Appendix D*

```
10  REM  ****************************************
20  REM  ****************************************
30  REM  PROGRAM: ANIMAL HEARTBEATS
40  REM  SUBJECT: LIFE SCIENCE
50  REM  PURPOSE: TO COMPARE THE RELATIVE SIZE OF
60  REM  ANIMALS AND THEIR HEARTBEAT RATES.
70  REM  PROGRAM USE: SIMULATION
80  REM  INTENDED AUDIENCE: LIFE SCIENCE STUDENTS
90  REM  PROGRAMMER: C. HEIMLER  6/15/86
100  REM  COPYRIGHT 1986
110  REM  ****************************************
120  REM  ****************************************
130 :
140  REM  VARIABLES
150 H$ = "HORSE":C$ = "CAT":P$ = "PIG":G$ = "GUINEA PIG":E$ = "ELEPHANT"
    :G$ = "GUINEA PIG"
160 B$ =  CHR$ (7): REM  BEEP SPEAKER FOR SOUND OF HEART BEAT
170  REM  AS$ = STUDENT'S ANSWER  A = NUMERIC VALUE OF AS$  HB = RANDOM
    INTEGER (1-5)  K$ = KEYBOARD CONTROL  Y = STORAGE FOR RANDOM INTEGER
     A$ = NAME OF ANIMAL

180  REM  R = NUMBER OF HEARTBEATS SOUNDED  X = VALUE FOR DELAY LOOP TO
    TIME RATE OF HEARTBEATS
182 :
184  ONERR  GOTO 1370: REM  SET SPEED TO NORMAL AND END PROGRAM
190 :
200  REM  PRINT TITLE
210  HOME : VTAB (9)
220  PRINT  TAB( 9)"**********************"
230  PRINT  TAB( 9)"*                    *"
240  PRINT  TAB( 9)"*                    *"
250  PRINT  TAB( 9)"*                    *"
260  PRINT  TAB( 9)"**********************"
270  SPEED= 25
280  VTAB 11: HTAB 11
290  PRINT "ANIMAL HEARTBEATS"
300  SPEED= 255
310  FOR P = 1 TO 2000: NEXT P
320 :
330  REM  PRINT LEARNING OBJECTIVE
340  HOME : VTAB (10)
350  SPEED= 100
360  PRINT "OBJECTIVE: TO COMPARE THE NORMAL RATE"
370  PRINT
380  PRINT "           OF HEARTBEAT FOR DIFFERENT"
390  PRINT
400  PRINT "           ANIMALS."
```

```
410  SPEED= 255
420  GOSUB 1500: REM  SPACE BAR CONTROL
430 :
440  REM  PRINT DIRECTIONS
450  HOME : VTAB (9)
460  SPEED= 100
470  PRINT "DIRECTIONS: STUDY THE HEARTBEAT RATES"
480  PRINT
490  PRINT "FOR THE ANIMALS IN THE TABLE THAT"
500  PRINT
510  PRINT "FOLLOWS."
520  SPEED= 255
530  GOSUB 1500: REM  SPACE BAR CONTROL
540 :
550  GOSUB 1390: REM   PRINT HEARTBEAT RATE TABLE
560  GOSUB 1500: REM  SPACE BAR CONTROL
570 :
580  REM  PRINT DIRECTIONS FOR HEARTBEAT SIMULATION
590  HOME : VTAB (7)
600  PRINT "I WILL SIMULATE THE NORMAL HEARTBEAT    OF AN ANIMAL."
610  PRINT
620  PRINT "BASED ON THE RATE OF THE HEARTBEAT YOU  DECIDE IF THE ANIMAL
     IS:"
630  PRINT
640  PRINT  TAB( 14)"1. ";H$
650  PRINT  TAB( 14)"2. ";C$
660  PRINT  TAB( 14)"3. ";G$
670  PRINT  TAB( 14)"4. ";P$
680  PRINT  TAB( 14)"5. ";E$
690  PRINT
700  PRINT "PRESS H FOR THE 'HEART' TO START         BEATING."
710  GET K$
720  IF K$ <  > "H" THEN 710: REM  CHECK ENTRY
730 :
740  REM  GENERATE 5 DIFFERENT RANDOM NUMBERS FOR SELECTION OF AN ANIMAL
     AND STORE IN AN ARRAY
750  FOR I = 1 TO 5
760 RN =  INT (5 *  RND (1) + 1)
770 HB(I) = RN
780  FOR J = 0 TO I - 1
790  IF HB(I) = HB(J) THEN 760
800  NEXT J
810  NEXT I
820 :
830  REM  COUNTER FOR SEQUENTIAL SELECTION OF ANIMAL FROM ARRAY
840 C = C + 1
850 :
860  REM   ASSIGNS NAME OF ANIMAL SELECTED TO A$ AND ASSIGNS VALUES FOR
    TIMING AND NUMBER OF HEARTBEATS
870  IF HB(C) = 1 THEN A$ = H$:R = 9:X = 155
880  IF HB(C) = 2 THEN A$ = C$:R = 18:X = 40
```

```
890  IF HB(C) = 3 THEN A$ = G$:R = 25:X = 20
900  IF HB(C) = 4 THEN A$ = P$:R = 15:X = 80
910  IF HB(C) = 5 THEN A$ = E$:R = 4:X = 220
920 :
930  GOSUB 1570: REM  BEEP SPEAKER TO SIMULATE ANIMAL HEART BEAT
940 :
950  REM  STUDENT'S SELECTION OF ANIMAL TO MATCH HEART BEATS
960  VTAB 4
970  PRINT "WHICH ANIMAL HAS A HEART THAT BEATS AT  THE RATE YOU JUST HE
    ARD?"
980  PRINT
990  PRINT  TAB( 15)"1. ";H$
1000  PRINT  TAB( 15)"2. ";C$
1010  PRINT  TAB( 15)"3. ";G$
1020  PRINT  TAB( 15)"4. ";P$
1030  PRINT  TAB( 15)"5. ";E$
1040  PRINT
1050  PRINT "PRESS THE NUMBER OF THE CORRECT ANSWER. THEN PRESS RETURN."

1060  VTAB 20: HTAB 19
1070  INPUT AS$
1080 A =  VAL (AS$)
1090  IF A < 1 OR A > 5 THEN  SPEED= 50: PRINT "PRESS A NUMBER FROM 1 TH
    ROUGH 5.  THEN  PRESS RETURN.": SPEED= 255: GOTO 1060
1100 :
1110  REM  CHECKS STUDENT'S ENTRY AS CORRECT OR INCORRECT
1120  HOME : VTAB (5)
1130  IF A = HB(C) THEN  PRINT "CORRECT ANSWER!": GOTO 1150
1140  PRINT "SORRY, WRONG ANSWER."
1150  PRINT
1155  IF HB(C) = 5 THEN  PRINT "YOU HEARD THE HEARTBEAT OF AN ELEPHANT."

1160  IF HB(C) < 5 THEN  PRINT "YOU HEARD THE HEARTBEAT OF A ";A$;"."
1170  PRINT
1180 :
1190  REM  MENU AND STUDENT SELECTION OF CHOICES
1200  PRINT "READY FOR ANOTHER TRY?"
1210  PRINT : PRINT
1220  PRINT  TAB( 4)"H - 'BEAT' ANOTHER ANIMAL HEART"
1230  PRINT
1240  PRINT  TAB( 4)"T - DISPLAY THE TABLE OF ANIMAL                HEARTBE
    ATS AGAIN"
1250  PRINT
1260  PRINT  TAB( 4)"Q -  QUIT THE PROGRAM"
1270  PRINT : PRINT
1280  PRINT  TAB( 4)"PRESS THE LETTER FOR YOUR CHOICE"
1290  GET K$
1300  IF K$ <  > "H" AND K$ <  >"T" AND K$ <  > "Q" THEN SPEED= 50: PRINT
    TAB( 4)"PRESS H, T, OR Q ":SPEED= 255: GOTO 1290
1310 :
1320  IF C = 5 THEN C = 0: REM  RESET COUNTER
```

```
1330 :
1340  IF K$ = "H" AND C = 0 THEN 740: REM     GENERATE A NEW SET OF RAND
    OM NUMBERS BEFORE SELECTING ANIMAL HEARTBEAT
1350  IF K$ = "H" THEN 840: REM  BEEP ANOTHER ANIMAL HEARTBEAT
1360  IF K$ = "T" THEN  GOSUB 1390: GOSUB 1500: GOTO 580
1370  SPEED= 255: HOME : END
1380 :
1390  REM  SUBROUTINE TO PRINT TABLE OF ANIMAL HEARTBEAT RATES
1400  HOME : VTAB (9)
1410  PRINT "ANIMAL            HEARTBEATS/MINUTE"
1420  PRINT "------            ----------------"
1430  PRINT "1. ";H$; TAB( 25)"32-44"
1440  PRINT "2. ";C$; TAB( 25)"110-130"
1450  PRINT "3. ";G$; TAB( 25)"260-300"
1460  PRINT "4. ";P$; TAB( 25)"60-80"
1470  PRINT "5. ";E$; TAB( 25)"25"
1480  RETURN
1490 :
1500  REM  SUBROUTINE TO USE SPACE BAR TO CONTINUE RUN OF PROGRAM
1510  VTAB 20
1520  PRINT  TAB( 5)"(PRESS SPACE BAR TO CONTINUE)"
1530  GET K$
1540  IF K$ <  >  CHR$ (32) THEN 1530
1550  RETURN
1560 :
1570  REM  SUBROUTINE TO BEEP HEARTBEATS
1580  HOME : VTAB 9
1590  PRINT  TAB( 12)"??????????????"
1600  PRINT  TAB( 12)"?? HEARTBEAT ??"
1610  PRINT  TAB( 12)"??????????????"
1620  POKE 34,14: REM    SET TOP MARGIN OF TEXT WINDOW TO PREVENT SCROLL
    ING OF LINES 1590-1610
1630  REM  HEARTBEAT RATE COUNTING AND JIMING LOOPS
1640  FOR T = 1 TO R
1650  FOR P = 1 TO X: PRINT "": NEXT P
1660  PRINT B$
1670  NEXT T
1680  TEXT : REM  RESET TEXT WINDOW
1690  HOME
1700  RETURN
```

Criteria for the Design and Selection of Educational Software

Appendix E

General

1 Effective and appropriate use is made of the computer.
2 Student interest is maintained throughout the program.
3 The program achieves its objectives.
4 The student can use the program without seeking assistance.
5 The level of difficulty (concepts and vocabulary) is compatible with the student user.
6 Any use of sound in the program can be turned off.
7 The cost of the program is appropriate in relation to the educational value.
8 Text used in the program is in good taste and free of slang, racial or ethnic slurs, sexism, and negative stereotypes.

Clarity

1 Objectives and purposes for the program are stated clearly.
2 Instructions for use of the program are clear and precise.
3 Clear and accurate external documentation is provided if needed.
4 Instructions are repeated if necessary.
5 Screen formats are neat and uncluttered.

Interactive

1 If appropriate, the program contains a menu.
2 The rate at which the program runs is controlled by the student.
3 Positive feedback is provided in response to correct student entries.
4 Response to errors is free of harsh or demeaning content.
5 Audible feedback in response to student errors can be turned off.
6 Failure is not rewarded with cute or clever graphics.
7 A summary of student performance is provided at the conclusion of the program.

8 The number of attempts allowed the student for entering an answer is not greater than three.

Accuracy

1 The program is free of programming errors.
2 Punctuation and grammar are correct.
3 Content is factually correct and up to date.

Table of ASCII Character Codes

Appendix F

ASCII character codes

ASCII Code	*Keystroke*	*ASCII Code*	*Keystroke*	*ASCII Code*	*Keystroke*
0	Ctrl @	34	"	68	D
1	Ctrl A	35	=	69	E
2	Ctrl B	36	$	70	F
3	Ctrl C	37	%	71	G
4	Ctrl D	38	&	72	H
5	Ctrl E	39	'	73	I
6	Ctrl F	40	(	74	J
7	Ctrl G	41	)	75	K
8	Ctrl H	42	*	76	L
9	Ctrl I	43	+	77	M
10	Ctrl J	44	,	78	N
11	Ctrl K	45	-	79	O
12	Ctrl L	46	.	80	P
13	Ctrl M	47	/	81	Q
14	Ctrl N	48	0	82	R
15	Ctrl O	49	1	83	S
16	Ctrl P	50	2	84	T
17	Ctrl Q	51	3	85	U
18	Ctrl R	52	4	86	V
19	Ctrl S	53	5	87	W
20	Ctrl T	54	6	88	X
21	Ctrl U	55	7	89	Y
22	Ctrl V	56	8	90	Z
23	Ctrl W	57	9	91	*[
24	Ctrl X	58	:	92	* \
25	Ctrl Y	59	;	93	]
26	Ctrl Z	60	<	94	^
27	Esc	61	=	95	*-
28	**n.a.	62	>	96	*`
29	Ctrl Shift M	63	?	123	*{
30	Ctrl ^	64	@	124	*|
31	**n.a.	65	A	125	*}
32	Space	66	B	126	*~
33	!	67	C		

* Not available on the Apple II keyboard.
** Not available on the Apple II, IIe, or IIc keyboard.

INDEX

A

Alphabetical sorts, 175–179
  with corresponding numbers, 178
AND, 50
Animal Heartbeats (computer program), 3, 5, 255–258
  formatting for title of, 80
  internal documentation for, 239
  learning objective in, 18
  outline of, 16
Animation, 158–162
  of character graphics, 186–189
  defined, 158
  in menus, 136
Applesoft BASIC, review of, 39–71
Arithmetic operations
  order of precedence for the Apple, 211
  order of precedence for BASIC, 214
Array(s), 58–60
  defined, 58
Arrow (created with characters), 183
ASC function, 65–66
ASCII (American Standard Code for Information Interchange) values, 51, 65–66, 123, 130, 131, 153, 176, 232, 261
Assignment statement(s), 41–48

B

Bank Street Writer, 4
BASIC
  Applesoft, 39–71
  structured, 25–29
Blastoff (computer program), 182
Bloom, Benjamin, 20
Borders, screen, 78, 96–98
Bottom margin, changing with POKE statements, 93, 94
Branching, 48–52
  defined, 48

C

Calling to the disk, 136
CALLs
  for cursor positioning, 82–84
  for screen formatting, 81–86
  for text clearing, 84–85
Caret (^), 90
Character graphics, 183–189
  animation of, 186–189

Character graphics (*continued*)
defined, 183
planning, 184–186
CHR$ (4), 135–136
CHR$ function, 65, 153
defined, 66
CLEAR statement, 107
Colon(s)
to force indentations, 238–239
to separate statements, 52, 90
Comma(s)
in numerals, 192–194
to separate strings, 45
Comma insertion subroutine, 192–194
Computer assisted instruction (CAI), 4
Concatenation, 64–65, 231
defined, 64
Conditional statements, 49–51
and tracing line numbers, 215, 216–217
CONTinue command, in debugging, 210–215
Conversion functions, 67–68
Counter(s), 43, 108, 143–150
digital, 179–182
initializing, 145
in subroutines, 146–147
Counter variable (I), 55
Counting
correct responses, 144–145
entries, 148–150
incorrect responses, 144
speed, 180
Criterion statement, 110
Cursor
positioning with CALLs, 82–84
in right justification, 201

D

Database programs, 4
DATA statement(s), 41, 43–45, 59, 107
containing both numeric and string data, 45
in decimal point alignment routine, 205
defined, 43
as source of programming errors, 220–226
Debugging, 54, 209–232
defined, 211
detecting errors with READ and DATA statements, 220–226
using ONERR GOTO statement, 226–232
using STOP and CONTinue, 210–215
using TRACE command, 215–220
Decimal point alignment routine, 203–207
Delay, 57
Delay loop(s), 75, 86–88, 161, 181
uses of, 87
Dialogue programs, 103–104
Digital counter, 179–182
Digital timer, 180–182
DIM statement(s), 60, 107
Disk, accessing separate programs on, 135
Division by zero, 228
Documentation, 233–242
defined, 234
external, 240–242
format for, 237–242
internal, 234–240, 241
minimal, 236
reasons for, 234–235
what to document, 236
Documentation guide, 242
Dollars and cents, formatting output for, 197–200
Dollars and cents conversion routine, 197–200
Drill and practice programs
defined, 2–3
values of, 103
Duplication check loop, 170–171

E

Educational program(s), 1-37
  advantages of, 4
  criteria for design and development of, 22-24
  criteria for design and selection of, 259-260
  defined and described, 2-4
  how to write (overview), 10-17
  incorporating user control into, 76-77
  interactive capability of, 102-106
  learning objectives in, 18-22
  minimal requirements of, 23-24
  optimum length of, 12-13
  outlining, 14-17
  planning, 10-12
  programming content of, 31-37
  selecting a topic for, 5-9
  teaching model for, 10-11
  techniques for adding variety and interest, 157-189
  techniques for making interactive, 101-138
  ways of including interaction, 102-105
Electronic spreadsheets, 4
End Punctuation (computer program), 3, 5, 75, 133, 152, 248-254
  outline of, 15
END statement(s), 44, 210
Entries, counting, 148-150
Equals (=) symbol ("assign," "replace"), 43
Error(s)
  detecting, 209-232
  DOS, 227
  logic, 212-214
  syntax, 212, 223-224, 227
  types of, 212-214
Error message(s), 54, 116, 212
  ?EXTRA IGNORED, 47-48
  ?OUT OF DATA ERROR, 44, 223, 230
  printing speed of, 117
  ?REENTER, 231
  ?SYNTAX ERROR, 223-224, 227
  ?TYPE MISMATCH ERROR, 42, 68
  ?UNDEF'D STATEMENT ERROR, 228
Evaluation of student achievement, 150-155
External documentation, 240-242
  example of, 242

F

Feedback, 3, 102-105, 140-143, 150-155
  how much, 154
  type of, 154
  visual, 105
Flashing mode, 91
  in menus, 136
FLASH statement, 78, 90-91
  defined, 90
Formal prompt(s), 111-113
FOR/NEXT loop, 34, 54-57, 119-120, 147, 221, 229
  advantage of, 55
  to assign values to a subscripted variable, 59
  in comma insertion subroutine, 193
  in decimal point alignment routine, 205-206
  delay loop, 86
  in digital counting programs, 180
  in numerical sorts, 172-175
  to print values stored in a subscripted variable, 59
  for programming a module, 27-28
  string functions in, 63
FOR/NEXT statement(s), 153
Frame(s), 74-78
  defined, 74
  display time of, 75
  first, 74-75

Frame(s) (*continued*)
last, 75
techniques for increasing visual interest of, 78

G

GET statement(s), 41, 48, 77, 103
defined, 121
differences between INPUT and, 48
and keyboard control, 129
in programming for multiple choice questions, 121–123
GOSUB statement(s), 28, 30, 51–52, 119, 185
defined, 51
in a multiple statement line, 53–54
with ONERR GOTO statement, 228
GOTO statement(s), 48–49, 217
limiting use of, 25, 26, 30
Graphics, character, 183–189

H

Hangman (computer program), 141
HOME statement, 75, 77, 81, 93, 97, 135, 186, 187, 188
defined, 74
Home Word (computer program), 4
HTAB, 27, 79–80, 159, 160, 162, 183

I

I (increment), 55, 56
IF-THEN statement(s), 49–51
to compare strings, 51
defined, 49
with relational symbols, 50
and tracing line numbers, 215, 216–217
Immediate execution mode, 211, 215
Indentations, forced with colons, 238–239
Indenting format as internal documentation, 239
Index, 58
Index variable of the loop, 55
Input(s), 114–116
advantage of accepting all as strings, 114–115
Input check(s), 13–14, 77, 116–121, 199
Input errors, handling, 116–120
Input question mark, 98–100
INPUT statement(s), 41, 46–48, 59, 98
in decimal point alignment routine, 205–206
defined, 46
differences between GET and, 48
and keyboard control, 129
Interaction, 3–4, 101–138
importance of, 102–106
Internal documentation, 234–240, 241
examples of, 237–240
International System (SI), formatting numeric output in, 192, 194
Inverse mode, 89, 90
in menus, 136
INVERSE statement, 27, 78
use of, 89–91
Inverse underscoring, 90

K

Keyboard control, 129-132
routine for, 130–131

L

Language, suggestions for educational software for, 7-8
Learning objective(s) (performance based), 18-22
  examples of, 19-20
  parts of, 19
  verbs used in, 19
LEFT$ function, 61-62, 63
  defined, 61
  with TAB and a FOR/NEXT loop, 64
Left margin, changing with POKE statements, 93, 94
LEN function, 66-67
  defined, 66
LET statement(s), 41-43
Line justification with TAB functions, 80
Line numbers, tracing, 215-220
Line renumbering program, 17
Line spacing with TAB functions, 80
List(s), 60
Literature, suggestions for educational software for, 7-8
Lock out, 122-123, 134
  defined, 122
Logical operators AND, OR, NOT, 50
Logic error(s), 212-214
Loops and tracing line numbers, 215
Lowercase letters, 126
  on screen, 91

M

Margins, changing with POKE statements, 93, 94
Mathematics, suggestions for educational software for, 8
Memory location, 69-70
Menu, 103
  defined, 132
  incorporating stimulants into, 136
  preferred programming technique for, 136-137
  programming, 132-138
  style of, 137
Menu driven program, advantages of, 133
Metric system (SI), formatting numerical output in, 192, 194
MID$ function, 62-64, 159, 160, 231-232
  in decimal point alignment routine, 204
  defined, 62
Module(s), 26, 27-28, 30
  defined, 27
  examples of, 27, 29
Money, formatting output for, 197-200
Motivational features, 154
Movement. *See* Animation
Multiple choice questions, programming to handle responses to, 121-123
Multiple statement line, 52-54, 89-90
  problem with, 54
  and tracing, 216

N

Negative reinforcements, 140-143, 154
  defined, 140
Nested loop, 55-56
NEW command, 165
NEXT statement, 221
Nim (computer game), 6
Normal mode, 89, 90
Normal speed, 164, 165
NORMAL statement(s), 27, 89, 91
NOT, 50
NOTRACE command, 215, 218
Null string, 42, 98
  checking for, 118

Number Machine, The (computer program), 5, 91, 109, 117, 221, 240, 243–247
  outline of, 14
  worksheet for, 247
Numerical sorts, 172–175
Numeric output, formatting, 191–207
  commas in numerals, 192–194
  decimal point alignment, 203–207
  dollars and cents, 197–200
  right justification, 200–203
  scientific notation conversion, 195–197
Numeric variable(s)
  difference between string variable and, 40
  examples of, 40
  used as counters, 144

O

ONERR GOTO statement, 226–232
ON GOSUB statement
  defined, 69
ON GOTO statement, 68–69
  defined, 68
OR, 50
Outlining a program, 14–17
Out of data error message, 44, 45, 223, 230
Output, numeric, 191–207

P

P (used as index variable), 57
Pause, 57
PEEK function, 152
PEEK statement, 69–70
  defined, 69
Percent correct, computer calculation of, 149
Plus sign (+), 65
Pointer, invisible, 222
POKE statement
  for canceling ONERR GOTO statement, 230
  for changing size of text window, 92–96, 97
  defined, 70
Positive reinforcements, 140–143, 154
  defined, 140
PRINT ASC, 66
PRINT CHR$, 65
PRINT statement(s), 34–35, 229
  containing a prompt, 46
  for programming menus, 133–135
  and screen appearance, 79
Program(s)
  accessing separate, 135–136
  Animal Heartbeats, 3, 5, 16, 18, 80, 239, 257–260
  End Punctuation, 3, 5, 15, 75, 133, 152, 250–256
  The Number Machine, 5, 14, 91, 109, 117, 221, 240, 243–249
Program design, 74–100
Program development, 74–100
Programming, 31–37
  in Applesoft BASIC, review of, 39–71
  four steps of, 31
  menus, 132–138
  structured, 25–30
  techniques, 74–100
  techniques for adding variety and interest, 157–189
  user control, 106
Program planning grid, 184–186
  illustration of, 185
  for planning character graphics, 184–186
Prompt(s), 46, 109–113, 179
  combination of formal and thematic prompts, 112
  defined, 109
  formal, 111–113
  purpose of, 111
  thematic prompt(s), 112–113

Q

Question mark (?) prompt, 46
Quit option, defined, 133
Quotation marks
  to designate null string, 42, 98
  to enclose strings, 40, 41, 45

R

Random function, 167–172
Random number generator, 167–172
  uses of, 168–169
  without repetition, 169–171
READ statement(s), 41, 43–45, 107
  in decimal point alignment routine, 205
  defined, 43
  as source of program errors, 220–226
Reinforcements, 140–143, 154
  negative, 140–143, 154
  positive, 140–143, 154
  sound as, 143
  use of, 141
  variety of, 140, 141–142
Relational symbols, 50
REMark statement(s), 32, 37
  to define functions of variables, 28, 30, 41
  for documentation, 234–240, 241
  to explain the functions of modules, 28, 30
  multiple statement lines with, 54
Responses, counting, 144–145
RESTORE statement, 107, 225
RESUME statement, 229
RETURN, 52, 228
Return to the menu option, 106–107
RIGHT$ function, 63
  defined, 62
Right justification, 200–203
  defined, 200
  example of, 201–202
RUN statement, 224–225

S

Sales tax, utility program to calculate, 198
Science, suggestions for educational software for, 8–9
Scientific notation conversion, 195–197
Screen border(s), 78, 96–98, 159–160
Screen display
  CALLs for formatting, 81–86
  formatting, 13, 73–100
  guidelines for good visual appearance of, 78–81
  techniques for adding variety and interest, 157–189
  techniques for increasing visual impact of, 78, 89–92
  visual impact of, 13
Semicolon(s)
  to create spaces, 89
  in cursor positioning CALLs, 84, 202
  to signal no line feed, 47
  trailing, 55
Significant digits, 196
Simulation programs, 104–105
  defined, 3
Single line statement(s), 26–27, 30
Slaying the Arithmetic Dragon (computer program), 154
Social studies, suggestions for educational software for, 8
Sorting routines, 172–179
  alphabetical sorts, 175–179
  numerical sorts, 172–175
Sound
  for evaluation, 151, 152
  as a reinforcement, 143
  for use in menus, 136
Spaces, formatting for in metric output, 192, 194
Space critter graphic program, 188

Speed
  changing, 164–167
  normal, 164, 165
SPEED command, 164–167
Spinner, 162–164
STEP, 56
STOP command, in debugging, 210–215, 219
STR$, 67, 68
  defined, 68
String(s), 42
  comparing with IF-THEN statements, 51
  defined, 40
  examples of, 40
  null, 42, 98
String functions, 61–64
String search routine, 104, 123–128
  search strategy, 124–125
String variable(s), 40–41, 42, 224
  difference between numeric variables and, 40
  examples of, 40
Structured BASIC, 25–30
  essential characteristics of programs, 25–26
Structured program, example of, 238
Structured programming, 25–30
  defined, 239
  guidelines for, 30
Student achievement, 139–155
  evaluation of, 150–155
Student workbook, 240–241
Submenus, 133
Subroutine(s), 26, 51–52
  advantages of, 128
  for comma insertion, 192–194
  counters and tallies in, 146–147
  defined, 51
  determining contents of, 32–33
  example of, 29
  with input checks, 119
  for scientific notation conversion, 195–197
  for search routine, 127–128
  for space insertion, 194
  and tracing, 217–219
  use of in structured programs, 28
Subscript, 58
Subscripted variables, 57–61
  double, 60–61
  multiple, 61
  requiring more than 11 boxes, 60
Substring(s), searching for, 125–126
Supplementary documentation, 240–242
Syntax error(s), 223–224
  defined, 212

T

TAB command, 183, 184
TAB functions, 35, 64, 99
  for creating character graphics, 183, 184
  in decimal point alignment, 203–204
  with LEFT$ function and a FOR/NEXT loop, 64
  and line justification and line spacing, 80
  and screen appearance, 79–80
Tallies, 143–150
  in subroutines, 146–147
Taxonomy of Educational Objectives Handbook I: Cognitive Domain (Bloom), 20–21
Teacher's guide, 241
Terminal statement, 110
Text
  guidelines for appearance of, 78–81
  increasing visual interest of, 89–92
Text clearing CALLs, 84–85
Text motion. *See* Animation
TEXT statement, 93, 95
Text window, changing size of, 78, 92–96
Thematic prompt(s), 112–113

THEN statement(s), 120
Top-down programming, 25, 30
  defined, 26
Topics for educational software
  examples of, 7–9
  guidelines for selecting, 5–6
  sources of, 6
Top margin, changing with POKE statements, 92, 94
TRACE command, 215–220
  defined, 215
Tracing line numbers, 215–220
Trailing blank, 159
Trailing semicolon, 55
Trailing space, 159
Tutorial programs, 103–104
  defined, 3

U

Uppercase letters, 126
User control, 76–77, 106–109
  messages for, 166
  programming, 106
  typical subroutine for, 106–109
  See also Interaction
User friendliness. *See* Interaction
User interaction. *See* Interaction

V

VAL, 67–68
  defined, 67
Value judgment(s), 150
Variable(s), 40–43
  assignment to, 41–48
  for counting correct responses, 144–145
  defining functions of in REM statements, 28, 30
  identifying and initializing, 31–32
  names of, 40–41
  numeric, 40, 144
  string, 40–41, 42
  subscripted, 57–61
Variable name(s), 40–41
  recognition of in Applesoft BASIC, 40–41, 213
Vector(s), 60
Vertical display, 83
Video display. *See* Screen display
VTAB, 27, 79–80, 159, 160, 163, 183

W

Word processing programs, 4